CARNEGIE-MELLON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

SPICE PROJECT

The Server Manual

28 August 1984

>maps>data>kjf> Shell V1.51 of 01-Feb84
>bob>Spice>Min> Shell V1.51 of 01-Feb84
KJF
BOB
>Pas>Spice>Cousin Shell V1
RCB>BOOT> Shell V1.51-May
PAS
RCB

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Projects Agency or the U.S. Government.

Table of Contents

1 The Authorization Server

1.1 Introduction

The Authorization Server has the responsibility of authenicating a user's identity by checking a login name and password against the list of recoginzed users and their passwords. This version of the Authorization server is only temporary. The temporary Authorization Server runs on a remote PERQ so that if that PERQ is down then the Authorization Server will not be running. This Authorization Server identifies each user with a unique id. A user may have owner or world access privileges. This is described in the *Introduction to the Spice User's Manual.* The Authorization Server that will be implemented in the future is described in the *Sesame: The Spice File System Manual* and will include group access privileges.

1.2 Type Definitions

The following type are defined in Authdefs.pas.

```
   Auth_Var_Size = 30;

   No_User   = 0;          { files owned by "nobody" }
   First_User = 1;         { first valid user }
   Max_Users  = 1023;

Type
   Auth_Var = String[Auth_Var_Size];

   User_ID  = No_User..Max_Users;   { must be "bit10" }

   PassType =  Long;            { a two word value }
                        { 4 chars for a password??? }

   UserRecord = record
       Name:       Auth_Var;    { Name of the user }
       UserID:     User_ID;     { The user ID of the user. }
       EncryptPass: PassType;    { The encrypted password. }
       Profile:    APath_Name;   { Path name of the profile file. }
       NameOfShell: APath_Name;   { Name of the Shell.RUN File. }
     End;

   Machine_Name = String[255];

   Check_Type = (Check_Login,       { user is logging in }
           Check_User);        { user is changing parameters }

   Logged_User = record          { one logged-in user }
       UserID     :User_ID;
```

```
        UserName    :Auth_Var;
        MachineName :Auth_Var;
      End;

    Logged_User_Array = array[0..0] of Logged_User;
    Logged_User_List  = ↑Logged_User_Array;

CONST
    Auth_Error_Base   = 5000;

    UserNameNotFound  = Auth_Error_Base + 1;
    PassWordIncorrect = Auth_Error_Base + 2;
    AuthPortIncorrect = Auth_Error_Base + 3;
```

1.3 Routines

The following functions and procedures are found in Authuser.pas.

InitAuth

Call:

```
procedure InitAuth(
                RPort     : Port)
```

Parameters:

RPort

ARunLoad initializes a process address space from RunFileName (or from a run file structure in memory if p is not nil), and optionally starts it executing.

LoginUser

Logs a user into the authentication server.

Call:

```
procedure LoginUser(
            ServPort      : Port;
            UserName      : Auth_Var;
            Password      : Auth_Var;
            MachineName   : Auth_Var;
          var UserAuthPort : Port;
          var UserRec      : UserRecord)
      : GeneralReturn
```

Parameters:

ServPort–The authentication server port

UserName–the name of the user to check

Password–password for the user

MachineName

UserAuthPort–Port returned to the user if the name and password match

UserRec–Is filled with the user information if the user name and password match

Results:

Success–Valid user, logged in

other–Invalid user

LogoutUser

Logs a user out.

Call:

```
        function LogoutUser(
                                ServPort      : Port)
              : GeneralReturn
```

Parameters:

ServPort–The port for the authentication server.

Results:

Success–AuthPortIncorrect

ConfirmUser

Checks the UserAuthPort to ensure that the user is logged in. If so, returns useful information about the user

Call:

```
        function ConfirmUser(
                   ServPort      : Port;
                   UserAuthPort  :  Port;
                 var UserId :        User_ID;
```

```
                    var UserMachineName:   Auth_Var)
                : GeneralReturn
```

Parameters:

ServPort–The port for the authentication server

UserAuthPort–the signature port

UserID–Returns the ID number for the user

UserMachineName–Returns the name for the machine that the user is logged on.

Results:

Success–if the user is valid

CheckUser

Verifies a user name/pasword pair and returns information about that user.

Call:

```
function CheckUser(
                ServPort :      Port;
                UserName :      Auth_Var;
                Password  :     Auth_Var;
              var UserRec  :    UserRecord)
        : GeneralReturn
```

Parameters:

ServPort–The authentication server port

UserName–user name to be checked

Password–password for the user

UserRec–Is filled with user information if the user name and password match.

Results:

Success–Valid user

other–Invalid user

ChangeUserParams

This function changes the parameters for a logged-in user.

Call:

```
function ChangeUserParams(
                ServPort       : Port;
                UserName     : Port;
                CurrentPassword:  Auth_Var;
                ChangePassword :  Boolean;
                NewPassword:     Auth_Var;
                NewProfile:      APath_Name;
                NewShell :       APath_Name)
        : GeneralReturn
```

Parameters:

ServPort–The port for the user's authentication server.

UserName–Name of the user whose information is to be changed

CurrentPassword–current password in the user's record

ChangePassword–if true, change password

NewPassword–new password to replace the current one

NewProfile–path name of the profile file for the user

NewShell–name of the shell to be stored in the user record.

Results:

Success–if the user was added or changed.

AuthPortIncorrect–If the user did not have the proper access rights to add or change a user.

GetUserName

Gets the user name corresponding to a User ID.

Call:

```
function GetUserName(
            ServPort      : Port;
            UserId :        User_ID;
         var UserName:       Auth_Var)
      : GeneralReturn
```

Parameters:

ServPort–Authentication Port Server

UserID–User ID

UserName–Returns the name for the user

Results:

Success

UserNameNotFound

ListLoggedInUsers

Returns all of the users currently logged in to this Authentication Server.

Call:

```
function ListLoggedInUsers(
            ServPort      : Port;
        var UserList    : Logged_User_List;
        var UserList_Cnt:     long)
       : GeneralReturn
```

Parameters:

ServPort–Authentication server port

UserList–Returns a list of user-ID-Machine

UserList_Cnt–Returns the number of users logged in.

Results:

Success

2 The Environment Manager

2.1 Introduction

The Environment Manager provides a language and process independent way of sharing directory searchlists and other variables among different processes executing on one machine.

The Environment Manager provides a message interface to define and retrieve environment variables. An environment variable is a named variable that has a type, a scope, and a set of values associated with it. There are two types: a string-valued variable and a searchlist. The values of the variables are kept as a variable-length array of strings. An environment variable has a scope associated with it, either local or global. A local variable is seen only by a single process. The local environment of the parent is copied at process creation time and is passed to the child process. Once the copy is made, any subsequest changes are not shared between parent and child. Only global variables can be shared between processes. Global variables are visible to all the processes that are served by the Environment Manager.

A String-valued variable simply allows one to store and retrieve arbitrary strings.

A Searchlist-valued variable is a list of directories to be searched when looking for a file. The entries in a searchlist are either directory names or names of other searchlists. On lookup, all references to searchlists are expanded (replaced by their contents) until the expanded searchlist consists only of directory names. An entry that is a reference to a searchlist contains the name of the searchlist followed by a colon (':') (and optionally a subdirectory name). This is the same syntax that is used by the file system to denote searchlists. It is possible to have both a local and global environment variable with the same name. In this case the local variable takes precedent just as in Pascal scope rules.

Ordinarily, if a local searchlist exists with the same name as a global searchlist, the local searchlist will be used. The one exception to this rule is that when a local searchlist contains its own name, that becomes a reference to the global searchlist with the same name – not a recursive reference to itself. This behavior is useful because it allows the system to specify a default search list for a given subsystem by name and for that subsystem to reference its path by that name. However, a user may then define a local searchlist with the same name to override the normal search list for that system. The user can then use the global definition from within the local definition, allowing the user to add directories to the normal search list for the subsystem.

Note that this applies even if the original searchlist is a global search list. For example, Accent searches for run files within the "Run:" search list and all other files within the "Default:" search list. A user who wants to make both searchlists the same can define the global "Run:" search list as "Default:". Then, each time the "Run:" search list is resolved, the sytem will actually search the "Default:" search list for the process asking for the search. Since the "Default:" search list is first resolved locally, this allows the one (global) definition of the "Run:" search list to refer to different "Default:" search lists depending on which process resolves it.

2.2 Definitions

The following definitions are found in EnvMgrDefs.pas.

At the shell level, Searchlist environment variable names are distinguished from string environment variable names by having a colon (':') as the last character of the name. This colon is NOT used by any of the Environment Manager user interface calls and is not returned by ScanEnvVariables.

```
{ Env_Variable: A list of environment entries, each of which is a string.

const
  Env_Element_Size     = 255;  { MaxString }
type
  Env_Element          = string[Env_Element_Size];
  Env_Element_Array    = array [0 .. 0] of Env_Element;     { hack }
  Env_Variable         = ↑ Env_Element_Array;

{ A Searchlist name embedded in a searchlist string is followed by
{ a Searchlist_Separator character.

const
  Searchlist_Separator  = ':';


{ Env_Var_Name:  The name string for an environment variable.
{     The syntax of the name is the same as for an arbitrary
{     entry name in the name server.

const
  Env_VarName_Size    = Entry_Name_Size;
type
  Env_Var_Name   = string[Env_VarName_Size];


{ Env_Var_Type:  The environment variable type values.

type
  Env_Var_Type = (
    Env_String,          { Values are lists of strings }
    Env_SearchList);      { Value is a search list }


{ Env_Var_Scope:  Flag specifying whether to find environment variable
{     in the local table, global table, or using the normal method
{     of local and then global.

type
  Env_Var_Scope = (
    Env_Normal,          { Use the normal lookup method }
    Env_Local,          { Refer to name in per-process table }
    Env_Global);          { Refer to name in global environment
                         variable table }
```

{ Env_Scan_List: A list of environment variable names, types, and scopes.

```
type
  Env_Scan_Record =
    record
    VarName      : Env_Var_Name;
    VarType     : Env_Var_Type;
    VarScope     : Env_Var_Scope;
    end;

  Env_Scan_Array     = array [0 .. 0] of Env_Scan_Record;  { hack }
  Env_Scan_List      = ↑ Env_Scan_Array;
```

{ Error return values for Environment Manager.

```
const
  Env_Error_Base     = 1600;

  EnvVariableNotFound = Env_Error_Base + 1;
  WrongEnvVarType     = Env_Error_Base + 2;
  BadSearchlistSyntax = Env_Error_Base + 3;
  SearchlistLoop      = Env_Error_Base + 4;
  FirstItemNotDefined = Env_Error_Base + 5;
```

2.3 Functions

The following functions are found in EnvMgrUser.pas.

GetEnvVariable

Returns the value of an environment variable. If the variable is a searchlist, this does not evaluate any contained searchlist references.

Call:

```
                    Function GetEnvVariable(
                              ServPort : Port;
                              Name     : Env_Var_Name;
                              SearchScope: Env_Var_Scope;
                          var Variable : Env_Variable;
                    var Variable_Cnt: long;
                    var VarType  : Env_Var_Type;
                    var ActualScope: Env_Var_Scope)
                  : GeneralReturn
```

Parameters:

ServPort–Connection to the Environment Manager for process.

Name–name of environment variable.

SearchScope–where to search:

Env_Global	global environment only
Env_Local	local environment only
Env_Normal	Search the local environment. If the variable is not there, search the global environment.

Variable–returns a pointer to the variable (a variable-length array of strings).

Variable_Cnt–returns the number of entries in variable.

VarType–returns type of environment variable: Env_Searchlist or Env_String.

ActualScope–returns where the variable was actually found (Env_Local or Env_Global).

Results:

Success

Environment Variable not found

SetEnvVariable

Enters a new environment variable. If the variable is a search list, checks for valid search list syntax and ensures that each entry ends in a directory separator ("/") or a searchlist terminator (":").

Call:

```
Function SetEnvVariable(
          ServPort  : Port;
          Name      : Env_Var_Name;
          VarType   : Env_Var_Type;
          VarScope  : Env_Var_Scope;
          Variable  : Env_Variable;
```

```
                        Variable_Cnt: long)
                    : GeneralReturn
```

Parameters:

ServPort–Connection to the Environment Manager for process.

Name–name of environment variable.

VarType–type of variable to enter:
Env_String or Env_Searchlist

VarScope–Where to enter the variable:

Env_Global	global environment
Env_Local	local environment

Variable–pointer to the variable (a variable-length array of strings).

Variable_Cnt–number of elements in variable. If the value is an empty array, the name is deleted.

Results:

Success

BadName–if search list name is null or if an entry is malformed. Null search list names are ignored.

ResolveSearchList

The ResolveSearchList call is used to resolve the value of an environment variable of type Env_SearchList, recursively expanding any environment variable references contained therein. If undefined names are encountered during the expansion, they are ignored and the expansion is continued. It is, however, an error if the evaluation results in an empty search list.

Call:

```
Function ResolveSearchList(
            ServPort : Port;
            Name : Env_Var_Name;
            FirstOnly : boolean;
        var Variable : Env_Variable;
        var Variable_Cnt : long;
        var FirstDefined : boolean)
    : GeneralReturn
```

Parameters:

ServPort–Port for process environment

Name–name of search list.

FirstOnly–if TRUE, only return the first item in the expansion. If FALSE, return all items in the expansion.

Variable–returns a pointer to the search list (a variable-length array of directory names).

Variable_Cnt–returns the number of entries in search list.

FirstDefined–returns TRUE if the first element in the expansion exists. Returns FALSE is it could not be resolved (it was a reference to a search list that did not exist).

Results:

Success

SearchListnotfound

WrongEnvVarType

SearchListLoop

ScanEnvVariables

Lists the defined environment variables by name.

Call:

```
Function ScanEnvVariables(
            ServPort  : Port;
            SearchScope: Env_Var_Scope;
            EnvScanList: Env_Scan_List;
          var EnvScanList_Cnt: long)
      : GeneralReturn
```

Parameters:

ServPort–Port for process environment.

SearchScope–Env_Global list global variables only
Env_Local list local variables only
Env_Normal list all local variables, and all global variables that are not hidden by local variables with the same names.

EnvScanList–returns the list of variable names, types, and scopes.

EnvScanList_Cnt–returns the number of entries in EnvScanList.

Results:

Success

CopyEnvConnection

Creates a new connection to the SearchList manager, copying all of the local variables belonging to the old connection.

Call:

```
Function CopyEnvConnection(
            ServPort : Port;
            OldConnection : Port;
          var NewConnection : port)
      : GeneralReturn
```

Parameters:

ServPort–Any port to the Environment Manager.

OldConnection–port designating parent connection. If it is NullPort, the new connection will have no local variables; otherwise, it will receive copies of all the local variables from OldConnection.

NewConnection–returns port for new connection.

Results:

Success

NoMoreConnections

EnvDisconnect

Destroys a connection to the Environment manager and all associated variables.

Call:

```
Function EnvDisconnect(
          ServPort : Port)
      : GeneralReturn
```

Parameters:

ServPort–port designating connection. The port is deallocated.

Results:

Success

3 The I/O System

3.1 Introduction

The IO (input/output) servers provide processes with the interface to access the IO devices on a PERQ workstation. A standard interface to all the servers presents a common mechanism for acquiring use of the device that the given server manages, for making IO requests to the server for using that device, and finally for relinquishing use of that device. For most of the IO devices the servers permit processes to manipulate the device at the lowest level of functionality (i.e., the actual hardware registers). This feature is provided by supplying a fairly open-ended definition for the parameters to the main IO request routine. It is felt that this type of control can best provide the flexibility needed by processes to make the device meet the needs of their particular applications. This does, however, place greater responsibility upon you to program the operation of the device correctly and efficiently.

The primary role of the server then is to manage the actual details of device interaction (including the necessary physical memory requirements), to provide a convenient means for processes to perform IO operations, to handle the allocation and deallocation of the device, and to attempt to ensure the integrity of the device by restricting certain operations and recovering the device when it appears to be hung.

Each IO server registers itself with its local Name Server using a Name/Port pair. The Name part of the registration includes the name of the local machine and thus identifies the server (and the corresponding device) with the machine. Thus a process which desires access to a device on a remote machine can expect to receive a port to that remote device server when the local Name Server broadcasts the request for the Name over the network.

3.2 Devices

The IO system provides access to all the IO devices except the disk and network. There is one device server for each of the supported IO devices. The supported IO devices are:

1. Floppy GPIB RS232A RS232B (PERQ2 workstation only) Speech

These devices are currently supported by a Z80 microprocessor in the PERQ which acts as an IO processor.

There is a single set of standard interface routines that provides you with access to all of the device servers. The way in which you select to which device server your request is being directed is to specify the appropriate port associated with that device server as the first parameter of one of the standard interface routines. Each device server has a special port registered with the Name Server that provides this unique access to the associated device server. The standard interface routines are found in IOUser.Pas in LibPascal, and IO system definitions for various types used in the parameters for the routines are found in IODefs.Pas in LibPascal.

3.3 Basic Routines

The user interface to an IO Server is provided through a small set of standard routines:

```
1) Procedure InitIO(RPort: Port );


2) Function IO_Version(
      ServPort : ServerNamePort
     ): String;


3) Function OpenIO(
     ServPort  : ServerNamePort;
   Var IOPort    : ServerIOPort;
     UserPort  : UserEventPort
    ): GeneralReturn;

4) Function CloseIO(
      IOPort : ServerIOPort
     ): GeneralReturn;


5) Function SyncIO(
      IOPort        : ServerIOPort;
      Command        : IOCommand;
      CmdBlk        : Pointer;
      CmdBlk_Cnt     : Long;
    Var DataBuf       : Pointer;
    Var DataBuf_Cnt    : Long;
      DataTransferCnt : Long;
      TimeOut       : Long;
    Var Status        : IOStatusBlk
     ): GeneralReturn;
```

These routines are discussed in Sections 2.10.

There are two other routines that will be implemented in a future release of Accent:

```
a) Procedure AsyncIO(
      IOPort        : ServerIOPort;
      Command        : IOCommand;
      CmdBlk        : Pointer;
      CmdBlk_Cnt     : Long;
      DataBuf       : Pointer;
      DataBuf_Cnt    : Long;
      DataTransferCnt : Long;
      TimeOut       : Long
     );
```

```
b) Function Event(
     EventMsg    : Msg;
   Var EventType  : IOEvent;
   Var DataBuf    : Pointer;
   Var DataBuf_Cnt : Long;
   Var Status     : IOStatusBlk
     ): GeneralReturn;
```

Except for 1) and b), each of these is actually a remote procedure call to a server and is implemented by sending a message to and receiving a reply from the server in order to accomplish the requested operation. The message passing interface is provided by modules created by the Matchmaker utility.

3.4 Modes of Interaction

Two modes of interaction - Synchronous and Asynchronous - are available for accessing a server to perform IO operations. The terms Synchronous and Asynchronous, as used here, do not imply anything about the type of data that the device is transferring (e.g., for RS232, it says nothing about whether the data is Bisync or Async). Synchronous and Asynchronous here refer to the style of user interaction with the server.

In the Synchronous mode, the user process makes IO requests through the SyncIO call. The user process is then blocked waiting for the reply message holding the results of the operation. This mode is useful if you do not want to deal with the details of the message passing system, since the Matchmaker generated interface modules handle the packing, sending, receiving, and unpacking of the messages implementing the remote procedure call. The Synchronous mode is also useful when the user process does want to block itself until a certain operation has been performed.

The Asynchronous mode provides a means to queue IO requests, using the AsyncIO call, without being blocked. This permits the user process to continue with other processing while the IO operation is being performed. You are permitted to queue up more than one IO request with the actual queue size limited by the particular server and the available message backlog limit on the server's port. When the server has completed a requested operation, it sends the results to you in a message as an asynchronous event which you must handle. Obviously, this now requires you to deal with the message system by doing an explicit Receive in order to obtain the event. Upon receiving an event message, you can then call the Event routine which simply unpacks the message and fills in the parameters of the Event call according to the message content. (Thus you do not need to know about the details of IPC message formats.) Currently the Asynchronous interface is not implemented.

3.5 Function of Basic Routines

This section discusses in more detail the standard routines and their parameters.

Except for the InitIO and Event routines, the first parameter to each of the basic routines is a port which is created and owned by the particular server and, thus, identifies the device server to which the user's request is directed. Each server owns two important ports - a ServerNamePort and a ServerIOPort - which provide the access to users. The ServerNamePort is registered with the Name Server and associated with a unique string name identifying the particular server.

The current existing IO servers are registered as:

> "[MachineName]GPIBServer"
> "[MachineName]FloppyServer"
> "[MachineName]RS232AServer"
> "[MachineName]RS232BServer"
> "[MachineName]SpeechServer"

where MachineName is the name of the machine upon which the device resides. This provides the means to access the devices on other machines across the network as well as local access.

User programs acquire send access to a ServerNamePort through a LookUp call to the NameServer. The ServerNamePort can only be used in the call to open the device or within the IO_Version call. The server grants access to the device to the OpenIO caller by returning its ServerIOPort. The ServerIOPort is then used as the first parameter in all other calls to the server. Some servers (e.g., the Z80 supported device servers) may grant exclusive use of the device and not permit it to be opened again until it has been closed; other servers may permit multiple access.

The InitIO call is not directed to the server. The InitIO procedure is created by Matchmaker and is part of the Matchmaker generated user interface module that implements (using the message system) the remote procedure calls. The user calls InitIO with the parameter ReplyPort which is a port in the user's space and to which the server is given send access for those remote calls (IO_Version, OpenIO, CloseIO, and SyncIO) that are implemented with a message send followed by an explicit receive. If the ReplyPort equals NullPort in the user call, then InitIO will allocate a port in the user's space for the reply messages for remote calls. The user must call InitIO just once and before the first actual remote call (which should be OpenIO) to a server.

The last parameter in the OpenIO call is the UserEventPort. This is a port owned by the user and to which the server will send asynchronous event messages. If the user specifies NullPort for the UserEventPort in his call to OpenIO, then the Asynchronous interface will not be enabled. This means that the server will ignore calls to AsyncIO and will not pass any other asynchronous events to the user. For now, the user should specify NullPort as the UserEventPort since the AsyncIO interface is not implemented.

The parameters for the actual IO calls are fairly straightforward. The CmdBlk, for instance, is a generic pointer to which the user can recast his own device specific command block pointer. See IODefs.Pas for the definition of this device specific command block, IOCmdBlk. CmdBlk_Cnt indicates the number of command bytes to which CmdBlk points. The CmdBlk is required to have a long integer occupying its first 2 words which can be set by the user to provide an ID tag for the IO command request. This tag is then copied into the first 2 words of the Status block when the response to the IO request is made to the user. The ID tag is mostly useful for matching up server responses to IO requests made through AsyncIO. The remainder of the CmdBlk is completely device specific. DataBuf points to a buffer for data transfers and

DataBuf_Cnt is the size of the buffer in bytes. The number of data bytes to transfer is indicated by DataTransferCnt.

DataTransferCnt is not a redundant parameter. DataBuf_Cnt is only used as an indicator of how many bytes pointed to by DataBuf are actually transmitted to/from the server in the message associated with the remote procedure call. Thus, for example, in a SyncIO call to read N data bytes, DataBuf should be set to Nil, DataBuf_Cnt should be 0, and DataTransferCnt should be N when the user invokes the remote procedure. Upon return from SyncIO and assuming the server successfully carried out the request, DataBuf will point to a buffer holding the N bytes of data. This buffer will have actually been created by the kernel when it handles the receipt of the server generated response message to the remote call for SyncIO. The kernel maps the data pointed to in the response message into the user's address space and ultimately the Matchmaker generated interface module will set DataBuf to point to that piece of memory.

TimeOut, when applicable to a given command, indicates how long to wait for the command to complete before giving up on it. In most cases, a value of -1 means to not wait, zero means to wait indefinitely, and a positive value means to wait that many clock ticks where a tick is approximately 1 microsecond. The Status block holds information about the success or failure of the IO command along with any available device status bytes. In the event of failure, the information in the Status block may also show how much of the command was performed before failure occurred. The set of IOCommands available will include those like IORead, IOWrite, IOReset, IOSense, etc., that are generally applicable to most devices as well as some that are device specific. Commands include those for data transfer and control, those for device control and configuration, and simple directives to the server.

3.6 Example of SyncIO for GPIB

In this section the service provided through the SyncIO call, along with a description of the parameters, is presented in the context of a specific example. Full definitions for new types are provided in an IODefs module found in Section 4. For GPIB the call and parameters are:

```
SyncIO(     IOPort          : ServerIOPort;
            Command         : IOCommand;
            CmdBlk          : Pointer;
            CmdBlk_Cnt      : Long;
        Var DataBuf         : Pointer;
        Var DataBuf_Cnt     : Long;
            DataTransferCnt : Long;
            TimeOut         : Long;
        Var Status          : IOStatusBlk
       ): GeneralReturn;
```

IOPort	As previously mentioned, this is the GPIB server port returned to the user upon successfully executing the OpenIO call.
Command	This is the user requested IO command. Valid commands for GPIB are:

IOSense	IORead
IODevRead	IOReset

IOWrite	IODevWrite
IOWriteEOI	IOWriteRegisters
IOFlushInput	IOSetBufferSize
IOReadHiVol	IOFlushOutput
IOWriteHiVol	

Each command will be discussed after presenting the rest of the parameters.

CmdBlk	This points to a record holding the CmdIDTag as well as device specific command bytes that are required for the command. The user's pointer to his IOCmdBlk for GPIB should be recast to a generic pointer. The definition for CmdBlk is given in Section 4.
CmdBlk_Cnt	The number of valid bytes in the CmdBlk. This is always at least 4 to account for the CmdIDTag.
DataBuf	This points to a buffer for the data which is to be sent or received. It is only used with the IORead, IOWrite, IOReadHiVol, IOWriteHiVol, and IODevWrite commands.
DataBuf_Cnt	The number of data bytes held in the buffer pointed to by DataBuf.
DataTransferCnt	The number of bytes to read/write to/from the data buffer.
TimeOut	The maximum number of clock ticks to wait before giving up on the command. A TimeOut value that is zero means to wait indefinitely. A TimeOut value of -1 means to not wait at all for some commands (e.g. IORead to get data from the ring buffer) and means to wait indefinitely for other commands (i.e., where to not wait would make no sense). A positive TimeOut value means to wait that many clock ticks for the IO operation to complete. A clock tick is approximately 1 microsecond. For some commands, the TimeOut is irrelevant and is ignored. In some cases, when a command times out, the server will issue a device reset automatically. This is done for those commands for which a message is sent to the Z80. Since the Z80 always gives an ACK/NAK for each command message, a time out would indicate that the device is hung. The only command that can free the device for subsequent commands is device reset.
Status	This shows the success or failure of the Command and, in either case, indicates how much of the command was performed. Also included is device status from the most recently issued IOSense command since the last device reset was issued. Obviously, the device status could be empty. Status will be defined by IOStatusBlk, IOSenseStatusBlk, and GPIBSenseStatus (see the definitions in Section 4). In the IOStatusBlk, HardStatus is status information provided by the device for the given Command. For IORead, for example, it is the error byte that is supplied with each character in the input ring buffer. SoftStatus is supplied by the server and provides a logical indication of the Command completion status. A list of the values used for SoftStatus is exported by the IODefs module. IOSuccess is the value returned in SoftStatus when the Command is successful. When command bytes are present, CmdBytesTransferred indicates how many of the bytes were actually transferred. DataBytesTransferred serves a similar function when the Command involves

data transfer. DeviceStatus for GPIB indicates the last available status from the internal registers of the TMS 9914 GPIB Controller chip which provides the interface to the GPIB. Refer to the Texas Instruments "TMS59914 GPIB Controller Data Manual" for a detailed description of this chip.

3.7 Commands for GPIB

In a number of the commands, the Z80 will wait until the data in the Z80's GPIB output ring buffer has been transmitted over the GPIB bus before actually initiating the command. Since some commands seize control and/or reconfigure the bus, it is necessary for the Z80 to wait until its GPIB output buffer has been drained before initiating those commands. This ensures the integrity of previous IOWrite commands which have sent output data for GPIB.

The standard TimeOut mechanism used for most commands is:

TimeOut <= 0 means wait indefinitely

TimeOut > 0 means wait TimeOut clock ticks

and commands that do time out are followed immediately by a server issued device reset.

SoftStatus is returned for each command and indicates success or the reason for failure. A list of SoftStatus codes with their meanings can be found in the IODefs module. A Command whose parameters are in error will be rejected and SoftStatus will indicate why. These types of errors will not be listed in the discussions below. Assuming no parameter errors, the standard values for SoftStatus for most GPIB commands will be:

IOSuccess	Command completed successfully.
IOTimeOut	Command did not complete within the TimeOut period.
IOUndefinedError	Command was NAKed by the Z80.

1. IOSense provides 10 bytes of status information from the Z80. Upon return, DeviceStatus in Status holds the count and the status bytes. The first 6 status bytes represent the register values in the TMS 9914 GPIB Controller chip at the time of the last Z80 GPIB interrupt and the remaining 4 bytes show values current with the issued IOSense. (These are shown in the GPIBSenseStatus record in Section 4.) Note that current values for IntStat0 and IntStat1 cannot be obtained since reading those registers dismisses the interrupts that the bit maps in those registers represent. (See TMS 9914 Data Book for more details.) The server also maintains a copy of these 10 status bytes and copies them into the DeviceStatus field of Status for subsequent IOCommands. The server will always zero its copy of DeviceStatus after a device reset. TimeOut for IOSense is standard. SoftStatus will be IOSucess, IOTimeOut, or IOUndefinedError.

2. IOReset puts the GPIB Controller into the idle state by issuing a device reset for GPIB to the Z80. This clears the Z80's GPIB input and output buffers (any data is discarded) and puts the TMS 9914 into the idle state by performing a Software Reset aux command. The TMS 9914 interrupt mode is reset for Data In and Data Out interrupts only, the Hdfa/Hdfe aux commands are disabled (in case they were previously set), and any data holdoff is released using Rhdf aux command. The PERQ workstation's GPIB input ring buffer is not affected. This command is also issued implicitly by the server for some of the other commands when they time out. TimeOut for IOReset is ignored since the device reset should never fail.

3. IOWriteRegisters is used to program the TMS 9914 registers. CmdBlk points to the user's IOCmdBlk which, for IOWriteRegisters, contains an array of GPIBWriteRegister elements. Each GPIBWriteRegister is a pair of bytes where the first byte indicates the TMS 9914 register and the second byte is the value to be written. CmdBlk_Cnt indicates the total number of bytes and, thus, must be even. TimeOut and SoftStatus are standard. CmdBytesTransferred shows the total number of bytes transferered.

4. IOFlushInput is used to suspend GPIB bus activity and extract all remaining GPIB input data held in the Z80. This is done by issuing a Tca aux command to the GPIB Controller chip to suspend bus activity and sending all data accumulated in the Z80 GPIB input ring buffer to the PERQ workstation. In addition, any byte that is being held in the controller chip's Data In register is removed and also sent to the PERQ workstation. All data returned is deposited in the workstation's GPIB input ring buffer and can be obtained by the user with the IORead command. Note that the Tca aux command is not issued by the Z80 until the Z80's GPIB output buffer has been drained. TimeOut and SoftStatus are standard.

5. IOFlushOutput flushes the Z80's GPIB output data ring buffer by waiting until all the data has been drained from the buffer. TimeOut and SoftStatus are standard.

6. IORead is used to extract data from the PERQ workstation's GPIB input ring buffer. This does not require any interaction with the Z80 and, thus, a time out does not result in a device reset. When the user makes the remote call, DataBuf should be Nil, DataBuf_Cnt 0, and DataTransferCnt should indicate the number of bytes to read. Upon return from the call, DataBuf will have been set to point to the buffer holding the data sent by the server (and may be Nil). Each character in the PERQ workstation input ring buffer has a status byte (the value of IntStat0 at the time of the Data In interrupt) associated with it. The server extracts characters and puts them into the DataBuf until either the DataTransferCnt is satisfied, a character's status byte shows an error, or no characters remain and the TimeOut expires. DataBytesTransferred is set to the number of characters returned in the DataBuf. SoftStatus is IOSuccess if the command succeeds completely. IONoDataFound is returned if no characters were found and the TimeOut expires. IOTimeOut is returned if the requested DataTransferCnt was not satisfied and the TimeOut expired. If the server finds a character with its status byte showing an error, it terminates further reading, puts the character into the DataBuf, sets HardStatus with the character's status byte, and sets SoftStatus to the appropriate error (IOCircBufOverflow or IOEndOfInput). For TimeOut, a value of 0 means wait indefinitely, -1 means don't wait, and > 0 means the obvious.

7. IOWrite sends data to the Z80 to be transmitted by the GPIB controller chip using Data Out interrupts. The Z80 buffers the data and sends an ACK when it has room in its buffer for another packet (i.e., 12 bytes is the most that can be sent in a single data packet) of data from the PERQ workstation. The server handles the user's IOWrite command by packaging the user's data pointed to by DataBuf into 12-byte packets and sending them to the Z80 as indicated above. (Obviously, IOWrite will be optimal when the DataBuf_Cnt is a multiple of 12.) Thus the actual transmission of the last of the data bytes onto the GPIB bus can only be verified by the user following up with an IOFlushOutput command or some other command that waits until the Z80 GPIB output buffer has been drained. TimeOut and SoftStatus are standard. DataBytesTransferred will indicate how many bytes were actually sent to the Z80 for transmission.

8. IOWriteEOI is the same as IOWrite except that the last data byte will be sent with the GPIB bus EOI line set. The user must take care not to send more output data until the Z80 GPIB output buffer drains–otherwise, EOI may be set on the wrong byte. Draining of the buffer can be verified explicitly with IOFlushOutput or implicitly with one of the other commands.
9. IOSetBufferSize is used to set the size of the physical buffer that is used for the IOReadHiVol and IOWriteHiVol commands. The default size is set to 1024 bytes when the device is allocated by the server in the OpenIO call. CmdBlk_Cnt should be 8 for this command.
10. IOReadHiVol reads data from the GPIB using a DMA channel and thus provides a high transfer rate. As with IORead, DataBuf is Nil and DataBuf_Cnt is 0 when the user makes the remote call and are set appropriately upon return. DataTransferCnt must be greater than 1 (since the Z80 DMA cannot handle a byte count of 1). TimeOut and SoftStatus are standard. DataBytesTransferred indicates the number of bytes actually read. IOReadHiVol may be programmed to terminate early if EOI is raised by the sending device. When this feature is enabled and occurs, a completion of IOEndofInput is returned for SoftStatus and DataBytesTransferred must be checked to determine the amount of data actually received.
11. IOWriteHiVol writes data to the GPIB using a DMA channel. DataBuf points to a buffer holding DataBuf_Cnt bytes. DataTransferCnt has the same restriction as for IOReadHiVol. TimeOut and SoftStatus are standard. DataBytesTransferred indicates the number of bytes actually written to GPIB and only needs to be checked when SoftStatus is not IOSuccess. The Z80 will wait for the Z80 GPIB output data ring buffer to drain before starting the HiVol operation.
12. IODevRead and IODevWrite are somewhat complex commands that, in their simplest form, can be used just to configure the GPIB bus to change the Talker/Listener device on the bus. The CmdBlk points to a record containing the CmdIDTag followed by a GPIBDevCmdBlk. See Section 4 for the definition of GPIBDevCmdBlk.

By setting all fields in GPIBDevCmdBlk to 0 or false except the PrimAddr and SecAddr, IODevRead would simply configure the bus with the device in PrimAddr and SecAddr as the new Talker; IODevWrite would configure a new Listener. If the SecAddr is not used, it should be set to 255. These commands are useful since they reduce a commonly used sequence of IOWriteRegister and IOWrite commands to a single command. They can also be used to configure the bus and set up the GPIB controller chip's interrupt mask registers appropriately for HiVol Read/Write commands. The basic algorithms for DevRead and DevWrite are given at the end of this section.

For IODevRead, CmdBlk_Cnt should be 10. If ReadCount in GPIBDevCmdBlk is non-zero, the Z80 will wait until ReadCount bytes have been read (using Data In interrupts) from the configured GPIB Talker device and the Z80 will then hold off further input using the Hdfa aux command. As the data accumulates in the Z80, it is sent up to the PERQ workstation's GPIB input ring buffer. The Z80 returns (and thus the IODevRead completes) only after the requested number of bytes have been read and sent to the PERQ workstation. To obtain the data, the user must subsequently use the IORead command (i.e., for IODevRead, DataBuf is NIL and DataBuf_Cnt is 0).

For IODevWrite CmdBlk_Cnt should be 9. DataBuf points to DataBuf_Cnt bytes of data to be transmitted on GPIB using Data Out interrupts. As with the IOWrite command, the data is shipped to the Z80 in packets. (In terms of efficiency, note that the first packet holds the 5 byte GPIBDevCmdBlk plus only 7 data bytes.)

For both IODevRead and IODevWrite, TimeOut and SoftStatus are standard. CmdBytesTransferred should be the same as CmdBlk_Cnt. DataBytesTransferred is valid only for IODevWrite and indicates the number of data bytes actually sent to the Z80 for transmission.

The Z80 will wait before initiating the DevRead and DevWrite commands until the Z80 GPIB output ring buffer has drained. The basic algorithm for each command is then given below using the following definitions:

AuxReg	- 9914 Auxiliary Command Register
DataOut	- 9914 Data Out Register
DataIn	- 9914 Data In Register
Mask0	- 9914 Interrupt Mask Register 0
Mask1	- 9914 Interrupt Mask Register 1

DevWrite:

```
With GPIBDevCmdBlk, GPIBDevCmdBlk.Options do

  begin
    AuxReg := Tca;
    AuxReg := Ton, off;
    AuxReg := Lon, off;
    If SetInt0Mask then Mask0 := Int0Mask;
    If SetInt1Mask then Mask1 := Int1Mask;
    If not OmitBusConfig then
       begin
         DataOut := Unt;
         If not OmitUnlisten then DataOut := Unl;
         If (0 <= PrimAddr) and (PrimAddr <= 30) then
           begin
             DataOut := Mla + PrimAddr;
             If (0 <= SecAddr) and (SecAddr <= 31) then
                 DataOut := Msa + SecAddr;
           end;
        { Note:  We load the DataOut with the first   }
        {        interface command (Unt) and the rest }
        {        are sent using Bus Out interrupts.   }
       end;

    AuxReg := Ton, on;
    If not OmitGoToStandby then AuxReg := Gts;
    {Now handle the data bytes }
    If DataTransferCnt does not equal 0 then
      begin
        DataOut := first data byte;
        { Remaining data bytes are transmitted using  }
        { Bus Out interrupts                          }
        If ForceEOI then last data byte is sent with EOI;
        If WaitOnData then
```

```
          wait till all data sent before returning
       else
          return without waiting;
     end;
  end { with };
```

<u>DevRead</u>:

```
With GPIBDevCmdBlk, GPIBDevCmdBlk.Options do
  begin
    AuxReg := Tca;
    AuxReg := Ton, off;
    AuxReg := Lon, off;
    If not OmitBusConfig then clear DataIn;
    If SetInt0Mask then Mask0 = Int0Mask;
    If SetInt1Mask then Mask1 := Int1Mask;
    If not OmitBusConfig then
      begin
        DataOut := Unt;
        If not OmitUnlisten then DataOut := Unl;
        If (0 <= PrimAddr) and (PrimAddr <= 30) then
          begin
            DataOut := Mta + PrimAddr;
            If (0 <= SecAddr) and (SecAddr <= 31) then
               DataOut := Msa + SecAddr;
          end;
        { Note: We load the DataOut with the first interface }
        {       command (Unt) and the rest are sent using    }
        {       Bus Out interrupts.                          }
      end;
    if HoldOffOnEOI then
       AuxReg := Hdfe, on
    else
       AuxReg := Hdfe, off;
    AuxReg := Hdfa, off;
    AuxReg := Rhdf;             {Release any previous holdoff }
    AuxReg := Lon, on;
    AuxReg := Gts;
    If ReadCount <> 0 then
      begin
        Wait until we have input ReadCount data bytes using
        Bus In interrupts and issue
        AuxReg := Hdfa, on before the last byte is input.
      end
  end { with };
```

3.8 Commands for RS232, Speech and Floppy

The valid SyncIO commands for the other servers will be discussed here. The description of the parameters to SyncIO is essentially the same as that given for GPIB in Section 2.4. Also, the description of th commands for GPIB in Section 2.5 is applicable except for the differences noted below for each device. The timeout mechanism, use of the status block, and the delay waiting for the device's output ring buffer to drain before initiating certain commands is similar to the case for GPIB.

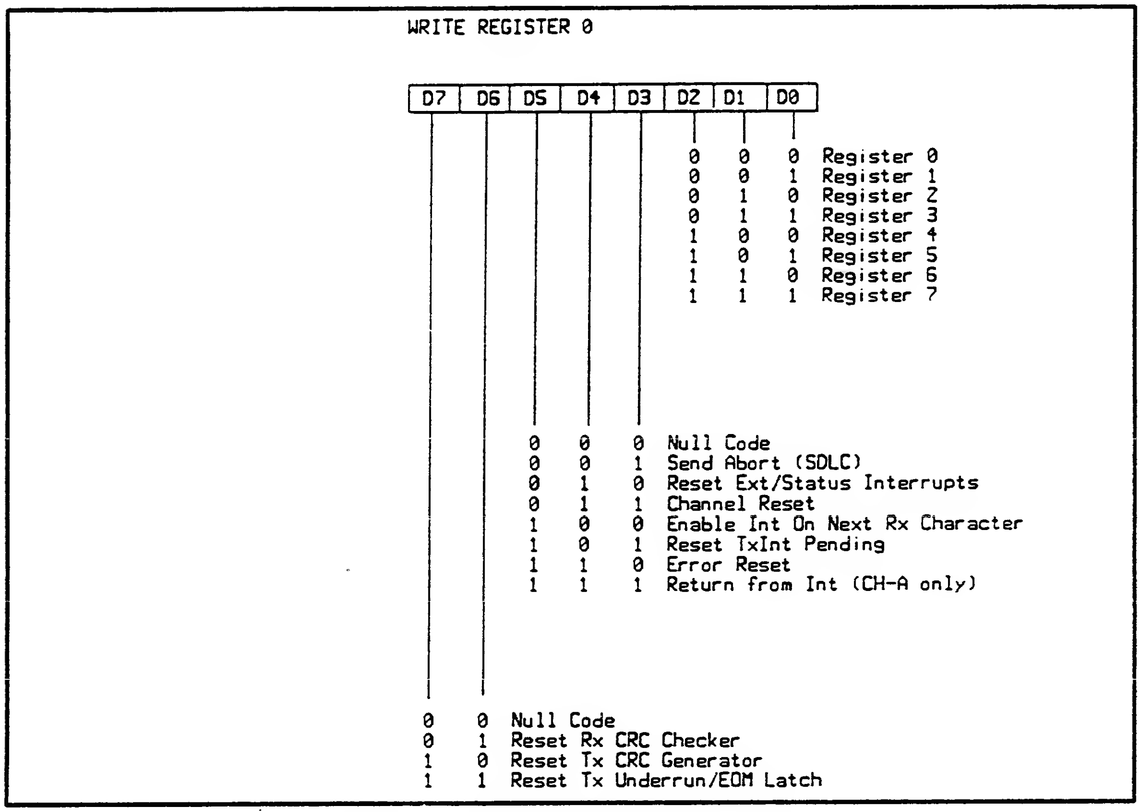
3.8.1 RS232A

The valid commands for RS232A are:

IOSense	IORead
IOReset	IOWrite
IOFlushInput	IOSetBufferSize
IOFlushOutput	IOReadHiVol
IOSetBaud	IOWriteHiVol
IOWriteRegisters	

IOSense	Only 2 bytes of status information are held in the DeviceStatus record. These correspond to the last available values obtained for the ReadRegisters 0 and 1 of the Serial IO (SIO) chip which implements the RS232 interface.
IOReset	This configures the RS232A channel to handle 9600 baud, 8-bit asynchronous data using 1 1/2 stop bits and no parity. The Data Terminal Ready (DTR) and Request To Send (RTS) signals (RS232 pins 20 and 4) are turned on and the SIO chip is programmed with Auto Enables mode. Auto Enables means that the Clear To Send (CTS) and Data Carrier Detect (DCD) inputs (RS232 pins 5 and 8) are used as the enable signals for the respective transmission and reception of RS232 data bytes. This means that CTS must be on before the SIO chip will transmit a byte. And the DCD signal must be on before the SIO chip will actually assemble data bytes from the incoming bit stream.
IOFlushInput	This simply sends all accumulated data held in the Z80 RS232A input ring buffer to the PERQ where it is deposited in the PERQ's RS232A input ring buffer.
IOSetBaud	This is used to modify the rate of the internal baud rate clock used for transmitting and receiving asynchronous RS232 data bytes in their bit-serial form. The baud rate codes to use are defined in IODefs.Pas and include a code (RSExt) for synchronous RS232 data where external clocks are used to synchronize the character bit stream. CmdBlk_Cnt should be 6 for IOSetBaud.
IORead	Similar to the GPIB example except that the per character status byte is the status byte provided by the SIO chip that accompanies each input data byte assembled by the chip. This byte is the value of SIO ReadRegister 1 which shows such errors as parity, framing, and overrun for the given input data byte as well as the end of frame condition and residue codes for SDLC data. Thus the SoftStatus code returned for IORead will be one of IOSuccess, IOTimeOut, IONoDataFound, IOCircBufOverFlow, IOParityError, IOFramingError, or IOEndOfFrame.
IOReadHiVol	Similar to GPIB. Note, however, that this command is not very useful for RS232 and that the EOI feature of GPIB is not present.
IOWriteRegisters	Permits you to program all SIO except WriteRegisters 1 and 2. These 2 registers are used to control the interrupt modes of the SIO chip. Registers 6 and 7 are

used to specify the sync characters used for synchronous RS232 data transfers. Bit definitions for the other 4 registers are given in Figures 1 through 4.


WRITE REGISTER 0

D2	D1	D0	
0	0	0	Register 0
0	0	1	Register 1
0	1	0	Register 2
0	1	1	Register 3
1	0	0	Register 4
1	0	1	Register 5
1	1	0	Register 6
1	1	1	Register 7

D5	D4	D3	
0	0	0	Null Code
0	0	1	Send Abort (SDLC)
0	1	0	Reset Ext/Status Interrupts
0	1	1	Channel Reset
1	0	0	Enable Int On Next Rx Character
1	0	1	Reset TxInt Pending
1	1	0	Error Reset
1	1	1	Return from Int (CH-A only)

D7	D6	
0	0	Null Code
0	1	Reset Rx CRC Checker
1	0	Reset Tx CRC Generator
1	1	Reset Tx Underrun/EOM Latch

Figure 1. Write Register 0 Bit Functions

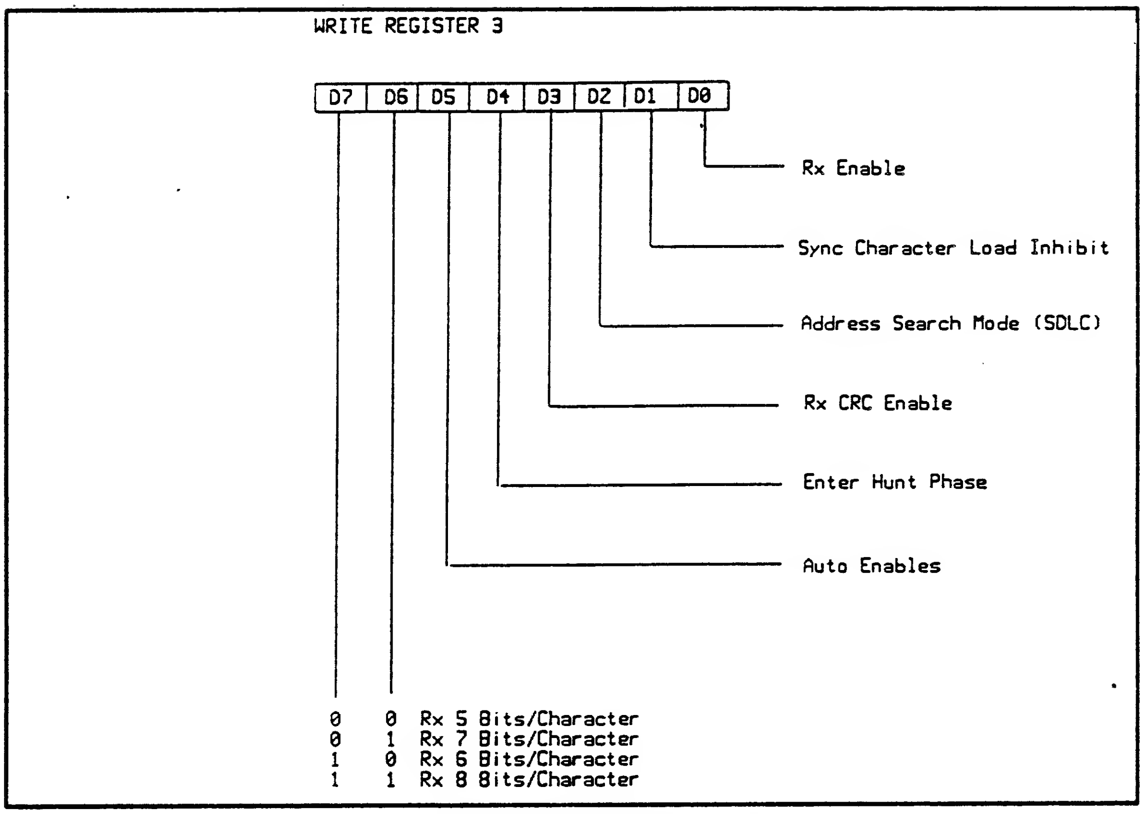


Figure 2. Write Register 3 Bit Functions

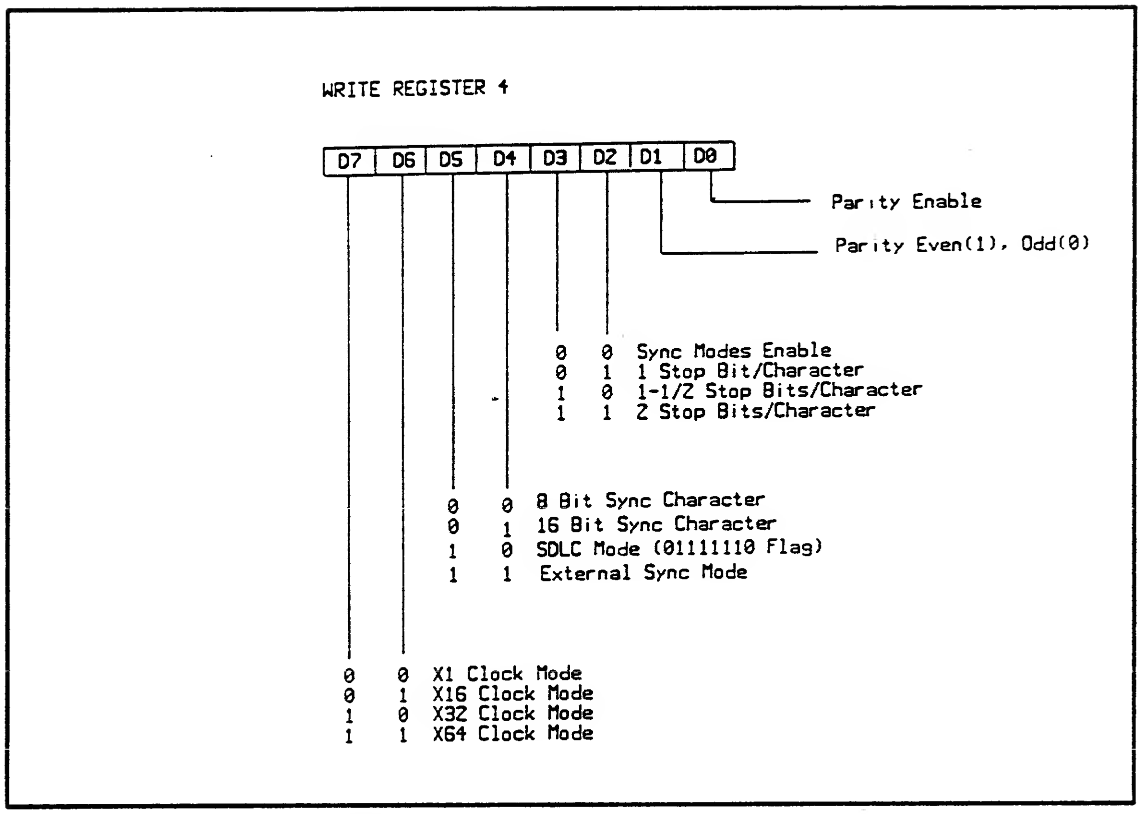


Figure 3. Write Register 4 Bit Functions

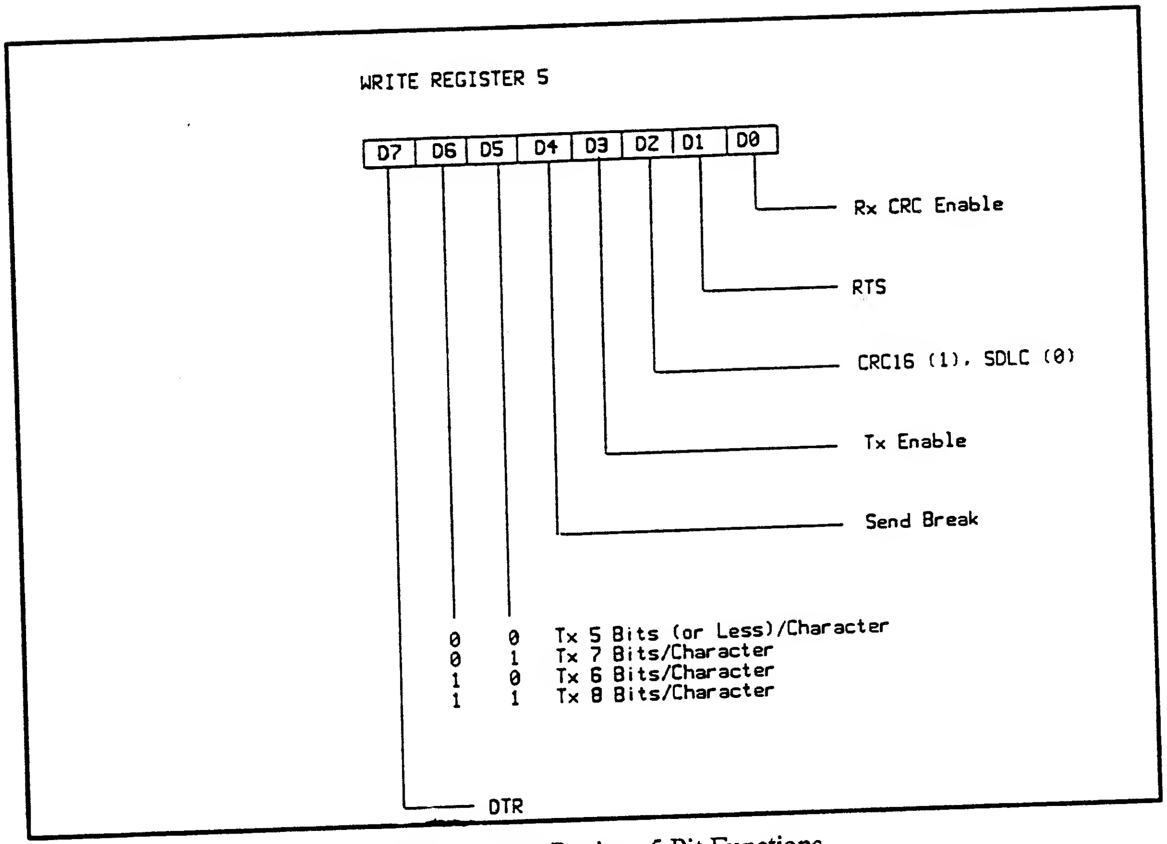


Figure 4. Write Register 5 Bit Functions

The byte masks used, for example, to program the chip for the default settings obtained by IOReset are:

Register #	Byte Mask (in octal)
0	030
0	100
4	110
3	341
5	352

3.8.2 RS232B

The description of RS232B is the same as RS232A except that the IOSetBufferSize, IOReadHiVol, and IOWriteHiVol commands are not present. Note again that RS232B does not exist on PERQ1 workstations.

3.8.3 Speech

The valid commands for speech are:

IOSense	IOWrite
IOReset	IOSetBufferSize
IOSetBaud	IOWriteHiVol
IOFlushOutput	

The description of these is the same as for RS232A except as noted below. The speech data output is implemented using the same type of SIO chip as for RS232 but configured instead for synchronous 8-bit data transmission using a single sync character of 252 (octal) which is automatically transmitted at the start of data transfers. The default baud rate is set at 32 KHz.

IOReset	Configures the hardware for the default state described above and clears the speech output ring buffer.
IOSetBaud	Changes the baud rate for bit-serial transmission of speech data to the actual speech output hardware in the PERQ. The SpeechTxRate field in the CmdBlk is set to obtain the desired bit rate in the following manner:

PERQ1:

$$\text{SpeechTxRate} = \frac{2.456 * 10\uparrow 6}{16 * (\text{Desired Baud Rate})}$$

PERQ2

$$\text{SpeechTxRate} = \frac{4 * 10\uparrow 6}{\text{Desired Baud Rate}}$$

A SpeechTxRate of 5 for PERQ1 and 125 for PERQ2 will give a baud rate of approximately 32 KHz. Note also that PERQs that have a portrait/landscape electromagnetic ranging tablet instead of a Summagraphics BitPad tablet have a further restriction. Since the electromagnetic ranging tablet shares the same baud rate clock with Speech, modifying the Speech baud rate may have unpredictable effects on the behavior of the tablet. You may have to unplug the tablet while your application runs, and you should restore the baud rate to 32KHz before your program terminates. This can easily be done by issuing the IOReset command.

3.9 Floppy

The valid commands for floppy are given in the table below, along with some other information that will be explained below.

Table 1. Valid Commands for Floppy

IO Command	CmdBlk Cnt	Returns Device Status	Returns Hard Status
IOReset	4	failure	on failure
IOSense	4	on success	no
IOSenseDrive	4	on success	no
IOReadID	6	on success	no
IOSetDensity	5	no	no
IORecalibrate	4	on failure	on failure
IOSeek	7	on failure	on failure
IOFormat	8	on failure	on failure
IORead	8	on failure	on failure
IOWrite	8	on failure	on failure

The parameters to SyncIO have a similar description as that for GPIB in Section 2.4 The main difference is in the Status parameter. For Floppy, the DeviceStatus record in the IOStatusBlk is only valid as indicated in the table above. From the table "on failure" that the command failed for reasons other than IOTimeOut, IOUndefinedError, or some parameter error. This means that the floppy controller hardware initiated the command but failed to complete it. The controller always provides status information at the termination of each command. These bytes are returned to you "on failure" as the DeviceStatus record which for floppy is a FloppyResultStatus record as defined in IODefs.Pas. Also, "on failure", HardStatus in Status will be assigned the first byte of the device status which is Status Register 0 from the controller and is represented in the FloppyResultStatus record as:

```
Unit        :  Bit2     {always 0}
Head        :  Bit1
NotReady    :  Boolean
EquipFault  :  Boolean
SeekEnd     :  Boolean
IntrCode    :  ( 00 - Normal
                 01 - Abnormal
                 10 - InvalidCmd
                 11 - DriveRdyChange)
```

DeviceStatus is also returned "on success" for those commands listed in the table whose function it is to explicitly extract status about the last floppy operation or current drive state. Also shown in the table is the CmdBlk_Cnt required for each of the floppy commands. From the definition of IOCmdBlk in IODefs.Pas, we see that the command bytes for the commands involving actual floppy IO include a device address specifying the Unit, Head, Cylinder, and Sector. The range of these parameters is:

Unit	- always 0
Head	- 0 ... 1
Cylinder	- 0 ... 76
Sector	- 1 ... 26

For read/write commands, the logical mapping of floppy sectors is done by first increasing the sector number, then the cylinder number, and lastly the head number. For IORead and IOWrite, the starting address on the floppy is given in the CmdBlk and DataTransferCnt indicates how many bytes should be read/written. To satisfy the requested number of bytes, the server will continue to read/write consecutively numbered sectors and will continue, if necessary, by advancing to the next cylinder on the same side. When the last cylinder on side 0 is read/written, it switches to side 1 at cylinder 0 and sector 1 and continues until the byte count is satisfied. Thus the server will implicitly do seek operations in order to satisfy the DataTransferCnt. If the DataTransferCnt given is greater than the number of bytes that could possibly be read/written from/to the floppy from the starting address to the end of the floppy, then the server will not attempt to start the command and will reject it with an error code of IONotEnoughData / IONotEnoughRoom.

The other floppy commands are fairly straightforward. The IOFormat command is used to format all 26 sectors of a single track designated by the address in the CmdBlk. You must also specify in the CmdBlk the default data pattern to be written into the formatted sectors of that track. DataBuf for IOFormat must point to a buffer holding 26 4-byte SectorIDs for each of the 26 consecutive sectors on the track. Each 4-byte SectorID is given as:

Byte	Information	Range
1	Cylinder	0...76
2	Head	0...1
3	Sector	1...26
4	SectorSizeCode	0...1

where a SectorSizeCode of 0 is used for single density with 128 bytes per sector and 1 is used for double density with 256 bytes per sector. The server does not check the 104 (26 x 4) byte buffer of SectorIDs for correctness. You can also perform hardware interleaving of floppy sectors by simply specifying the desired sector number in each SectorID. Thus if the list of sector numbers specified in the 26 consecutive SectorIDs is given by 1, 14, 2, 15, 3, 16, . . ., 12, 25, 13, 26, then a hardware interleave factor of 2 is achieved on that track. The IORecalibrate command forces the floppy to seek to head 0 and cylinder 0. IOReset does a device reset followed by a recalibrate. The timeout mechanism for floppy is the same as that described for GPIB in Section 2.5.

3.10 Definitions

The following definintions can be found in the Module IODefs.

```
const

 { Define return values for IO errors.

 IOBaseMsgID      = 4000;

 IOSuccess       = SURESS;
```

```
IOErr            = IOBaseMsgID;
IOUndefinedError      = IOErr + 1;
IOTimeOut           = IOErr + 2;
IODeviceNotFree       = IOErr + 3;
IOInvalidIOPort      = IOErr + 4;
IOBadUserEventPort     = IOErr + 5;
IOBadPortReference     = IOErr + 6;
IOIllegalCommand      = IOErr + 7;
IOBadCmdBlkCount       = IOErr + 8;
IOBadDataByteCount     = IOErr + 9;
IOBadRegisterNumber     = IOErr + 10;
IONotEnoughRoom       = IOErr + 11;
IOBadBaudRate        = IOErr + 12;
IONoDataFound        = IOErr + 13;
IOOverRun          = IOErr + 14;
IOParityError        = IOErr + 15;
IOFramingError        = IOErr + 16;
IOCircBufOverFlow      = IOErr + 17;
IOEndOfFrame         = IOErr + 18;
IOEndOfInput        = IOErr + 19;
IOBadSectorNumber      = IOErr + 20;
IOBadCylinderNumber     = IOErr + 21;
IOBadHeadNumber       = IOErr + 22;
IOUndeterminedEquipFault = IOErr + 23;
IODeviceNotReady       = IOErr + 24;
IOMissingDataAddrMark    = IOErr + 25;
IOMissingHeaderAddrMark  = IOErr + 26;
IODeviceNotWritable     = IOErr + 27;
IOSectorNotFound       = IOErr + 28;
IODataCRCError       = IOErr + 29;
IOHeaderCRCError       = IOErr + 30;
IOBadTrack          = IOErr + 31;
IOCylinderMisMatch      = IOErr + 32;
IODriveReadyChanged     = IOErr + 33;
IONotEnoughData       = IOErr + 34;
IOBadBufferSize       = IOErr + 35;

type
 ServerNamePort = Port;
 ServerIOPort  = Port;
 UserEventPort  = Port;

 IOCommand    = (IOSense,     IOReset,     IOWriteRegisters,
            IOFlushInput,  IOFlushOutPut, IORead,
            IOWrite,     IOWriteEOI,   IOReadHiVol,
            IOWriteHiVol,  IODevRead,    IODevWrite,
          IOSetBaud,      IOSetStream,  IOSetAttention,
          IOAbort,        IOSuspend,    IOResume,
          IOSeek,         IORecalibrate, IOFormat,
          IOReadID,       IOSenseDrive, IOSetDensity,
          IOSetBufferSize, IONullCmd);

IOEvent      = (IOReply, AsyncData, Attention, Distress, Acknowledge);

pIOMessage   = ↑IOMessage;
IOMessage    = record
           Head : Msg;
           Body : array[0..1023] of integer;

               { Note: The size of the Body is dependent
               {    upon the maximum size of messages
               {    that are passed between the Client
               {    and Server. This size is determined
               {    by the number and type of the parameters
               {    in the remote calls, the IPC conventions
               {    for packing data in-line, and the added
               {    parameters in the message that MatchMaker
               {    inserts for coded return values.
               {    The actual size needed for IO messages
               {    is 46 given current IPC conventions,
               {    MatchMaker requirements, and actual
               {    parameters for the remote calls. Future
               {    changes to any of the above may require
               {    modifying the size for the Body. Accent
               {    gurus say that 1024 is a reasonably safe
               {    maximum for the time being and that we
               {    should just use that here.

          end;

GPIBWriteRegister = packed record
            case RegNum: Bit8 { really 0..6 } of
             0: ( { To be defined later! } );
             1: (RegVal : Bit8)
           end;

SIOWriteRegister = packed record
            case RegNum: Bit8 { really 0..7 } of
             0: ( { To be defined later! } );
             1: (RegVal : Bit8)
          end;

GPIBDevCmdHead = packed record
```

```
            Options: packed record  { Bitmap to select cmd actions }
                    SetInt0Mask   : boolean;
                    SetInt1Mask   : boolean;
                    OmitBusConfig : boolean;
                    OmitUnListen  : boolean;
                    case boolean of
                      true : ( { for IODevRead cmd }
                            HoldOffOnEOI   : boolean;
                            unused2        : Bit3
                           );
                      false: ( { for IODevWrite cmd }
                            OmitGoToStandby : boolean;
                            WaitOnData     : boolean;
                            ForceEOI       : boolean;
                            unused1        : Bit1
                           );
                    end;
            Int0Mask  : Bit8;  { Mask for 9914 Interrupt Reg 0 }
            Int1Mask  : Bit8;  { Mask for 9914 Interrupt Reg 1 }
            PrimAddr  : Bit8;  { Primary Address of device }
            SecAddr   : Bit8;  { Secondary Address of device }
            case boolean of
              true:  ( { for IODevRead }
                    ReadCount : Bit8
                   );
              false: ( { for IODevWrite }
                    { nothing }
                   );
          end;
DensityType    = (SingleDensity, DoubleDensity);

pIOCmdBlk = ↑IOCmdBlk;
IOCmdBlk  = packed record
        CmdIDTag: long;
        case integer {IODevice} of
          0: {No Device for generic access}
            (case integer of
              0: {Byte access}
                (CmdByte: packed array[0..0] of Bit8);

              1: {Word access}
                (CmdWord: packed array[0..0] of integer);

              2: {Register access}
                (WriteReg: packed array[stretch(0)..stretch(0)] of
                           packed record
```

```
                  RegNum: Bit8;
                  RegVal: Bit8
                 end)
     );


  1: {GPIB}
     (case IOCommand of
       IOSetAttention: { AsyncIO only }
         (GPIBEnableATN: boolean);

       IOSetStream: { AsyncIO only }
         (GPIBEnableStream: boolean;
          GPIBBlockingFactor: integer);

       IOSetBufferSize:
         (GPIBBufferSize: long);

       IOWriteRegisters:
         (GPIBWriteReg: packed array[stretch(0)..stretch(0)]
                          of  GPIBWriteRegister);

       IODevRead, IODevWrite:
         (GPIBDevCmdBlk: GPIBDevCmdHead)
     );
  2: {RS232A, RS232B}
     (case IOCommand of
       IOSetAttention: { AsyncIO only }
         (RS232EnableATN: boolean);

       IOSetStream: { AsyncIO only }
         (RS232EnableStream: boolean;
          RS232BlockingFactor: integer);

       IOSetBufferSize: { RS232A only }
         (RS232BufferSize: long);

       IOSetBaud:
         (RS232TxBaud: Bit8;
          RS232RxBaud: Bit8);

       IOWriteRegisters:
         (RS232WriteReg: packed array[stretch(0)..stretch(0)]
                          of  SIOWriteRegister)
     );
```

```
      3: {Speech}
        (case IOCommand of
          IOSetAttention: { AsyncIO only }
            (SpeechEnableATN: boolean);

          IOSetBufferSize: { SpeechA only }
            (SpeechBufferSize: long);

          IOSetBaud:
            (SpeechTxRate: integer);

          IOWriteRegisters:
            (SpeechWriteReg: packed array[stretch(0)..stretch(0)]
                              of  SIOWriteRegister)
        );
      4: {Floppy}
        (case IOCommand of
          IOSetAttention: { AsyncIO only }
            (FloppyEnableATN: boolean);

          IOSetDensity:
            (FloppyDensity: DensityType);

          IORead, IOWrite, IOFormat, IOSeek, IORecalibrate,
          IOReadID, IOSenseDrive:
            (FloppyUnit  : Bit8;
             FloppyHead  : Bit8;
             case IOCommand of
               IORead, IOWrite, IOFormat, IOSeek:
                 (FloppyCylinder: Bit8;
                  case IOCommand of
                    IORead, IOWrite:
                      (FloppySector: Bit8);
                    IOFormat:
                      (FloppyFmtData: Bit8)
                 )
            )
        )
    end;

GPIBSenseStatus = packed record  { IOSense to GPIB provides }
            { first 6 bytes are status at time of last interrupt }
            IntStat0    : Bit8;  { Interrupt Status 0 }
            IntStat1    : Bit8;  { Interrupt Status 1 }
            IntAddrStat : Bit8;  { Address Status }
            IntBusStat  : Bit8;  { Bus Status }
            IntAddrSwch : Bit8;  { Address Switch }
            IntCmdPass  : Bit8;  { Command Pass Through }

            { next 4 bytes are current status  }
            CurAddrStat : Bit8;  { Address Status (now) }
            CurBusStat  : Bit8;  { Bus Status (now) }
            CurAddrSwch : Bit8;  { Address switch (now) }
            CurCmdPass  : Bit8;  { CurCmdPass (now) }
          end;

SIOSenseStatus = packed record
           { Read General Status - SIO Chip's Read Register #0 }
           RxCharAvailable  : boolean;
           IntPending       : boolean;
           TxBufferEmpty    : boolean;
           DCD              : boolean;
           SyncHunt         : boolean;
           CTS              : boolean;
           TransmitUnderRun : boolean;
           BreakAbort       : boolean;

           { Read Special Condition - SIO Chip's Read Register #1 }
           AllSent          : boolean;
           Residue          : Bit3;
           ParityError      : boolean;
           RxOverRun        : boolean;
           CrcFramingError  : boolean;
           EndOfFrame       : boolean;
         end;

FloppyResultType   = (NoStatus,   {no status available}
             HeadChange,  {status from Seek or Recalibrate}
             DriveSense,  {status from SenseDrive}
             CmdResults   {status from other drive cmds}
           );

FloppyResultStatus = packed record
             { Floppy device status is always returned as part  }
             { of the result phase of a cmd to the controller.  }
             { Z80 simply maintains a copy of the status results}
             { for the last cmd.                                }
             StatusType: FloppyResultType;
             Unused    : Bit6;
             case FloppyResultType of
               NoStatus: ({nothing});
            HeadChange, DriveSense, CmdResults:
             (Unit: Bit2;
              Head: Bit1;
              case FloppyResultType of
                DriveSense:
                  (TwoSided     : boolean;
                   AtTrack0     : boolean;
                   DriveReady   : boolean;
                   WriteProtected: boolean;
                   DriveFault   : boolean
                  );
                HeadChange, CmdResults:
                  (NotReady  : boolean;
                   EquipFault: boolean;
                   SeekEnd   : boolean;
                   IntrCode  : (Normal, Abnormal,
                                InvalidCmd, DriveRdyChange);
                   case FloppyResultType of
                     HeadChange:
                       (PresentCylinder: Bit8);
                     CmdResults:
                       (NoAddrMark    : boolean;
                        NotWritable   : boolean;
                        NoData        : boolean;
                        Unused1       : Bit1;
                        Overrun       : boolean;
                        DataError     : boolean;
                        Unused2       : Bit1;
                        TrackEnd      : boolean;
                        NoDataAddrMark: boolean;
                        BadTrack      : boolean;
                        ScanFail      : boolean;
                        ScanHit       : boolean;
                        WrongCylinder : boolean;
                        DataCRCError  : boolean;
                        ControlMark   : boolean;
                        Unused3       : Bit1;
                        CylinderID    : Bit8;
                        HeadID        : Bit8;
                        SectorID      : Bit8;
                        SectorSizeCode: Bit8
                       )
                  )
             )
          end;
IOSenseStatusBlk = packed record
```

```
            StatusCnt: Bit8;
            case integer {IODevice} of
              1: {GPIB}
                (GPIBStatus: GPIBSenseStatus);
              2: {RS232A, RS232B, Speech}
                (SIOStatus: SIOSenseStatus);
              3: {Floppy}
                (FloppyStatus: FloppyResultStatus);
              0: {otherwise} { for generic access }
                (case integer of
                  1: (StatusByte: packed array[stretch(1)..stretch(1)]
                      of Bit8);
                  2: (StatusWord: packed array[stretch(1)..stretch(1)]
                      of integer);
                  3: (SByte    : packed array[stretch(1)..stretch(12)
                      ] of Bit8)
                )
            end;

  IOStatusBlk    = packed record
            CmdIDTag            : long;
            HardStatus          : integer;
            SoftStatus          : integer;
            CmdBytesTransferred : long;
            DataBytesTransferred : long;
            DeviceStatus        : IOSenseStatusBlk;
           end;

const
  {}
  { Baud rate codes for RS232.
  {}
  RSExt      = 0;
  RS110      = 1;
  RS150      = 2;
  RS300      = 3;
  RS600      = 4;
  RS1200     = 5;
  RS2400     = 6;
  RS4800     = 7;
  RS9600     = 8;
  RS19200    = 9;  { only for RS232A }
```

3.11 Routines

The following Routines are from IOUser.pas.

InitIO

InitIO is used to specify the port in the calling process's port space that will be used by the other interface routines in IOUser.Pas to accomplish the remote procedure calls to the IO servers. The port specified serves as the reply port upon which the response from the server will be received. This procedure should be called once and before any of the other routines in IOUser.Pas are used.

Call:

Procedure InitIO(RPort: Port)

Parameters:

RPort– This is a port in your process's port space. If NullPort is specified, then InitIO will create a new port. Most programs find it convenient to just specify NullPort.

Results:

None.

IO_Version

This function returns a string which identifies the version number of the server.

Call:

```
Function IO_Version(
          ServPort : ServerNamePort)
     : String
```

Parameters:

*ServPort–*This is the port that identifies the IO server to which your request is being directed. It is the port that you obtain from the Name Server when you perform a LookUp to the Name Server for the desired IO server.

Returns:

The server's version number in string form.

OpenIO

Open IO is used to acquire the right to use the device managed by the selected IO server. The server grants access to a process by returning a port which the process then uses in the other IO calls. Some servers grant exclusive use to a single process and thus OpenIO performed by other processes will be rejected until the current process relinquishes access with the CloseIO call. OpenIO will always enable the Synchronous interface to the server and will only enable the Asynchronous interface if a UserEventPort is specified. Currently, the Asynchronous interface is not implemented and thus NullPort should be used here.

Call:

```
Function OpenIO(
                ServPort  : ServerNamePort;
            Var IOPort    : ServerIOPort;
                UserPort  : UserEventPort )
          : GeneralReturn
```

Parameters:

ServPort–This is the port that identifies the IO server to which your request is being directed. It is the port that you obtain from the Name Server when you perform a LookUp to the Name Server for the desired IO server.

IOPort–This is the port returned by the server (if the OpenIO request is granted) which the caller uses to make subsequent IO requests.

UserPort–This is the port upon which the calling process wishes to receive Asynchronous events. This port should be set to NullPort until the AsyncIO call is implemented. NullPort directs the server to not enable the Asynchronous interface.

Results:

IOSuccess–if the request is granted.

IODeviceNotFree–if an exclusive-use device is already allocated.

CloseIO

CloseIO is called when you are through using the device managed by the selected server. The specified ServerIOPort identifies to which server the CloseIO is being directed.

Call:

```
Function CloseIO(
                 IOPort : ServerIOPort )
           : GeneralReturn
```

Parameters:

IOPort–This is a ServerIOPort that uniquely identifies the server to which the CloseIO is directed. The port to use here is the ServerIOPort returned by the server in the OpenIO call.

Results:

IOSuccess-if no errors; otherwise returns an identifying error code.

SyncID

SyncIO is the primary call to be used in performing actual IO operations on the device managed by the selected IO server. The specified ServerIOPort identifies the server to which the request is directed. The other parameters are specified according to the particular requirements for the device and the valid set of commands applicable to the device. These are discussed in more detail in other parts of this document and are also presented in the module IODefs.Pas. The server performs the operation and the SyncIO call returns whenever the operation completes or times out. A device dependent status block is returned which shows the degree of success in the completion of the operation. The set of IO error codes as well as the status block definition can also be found in IODefs.Pas. In most cases the server will automatically issue a device reset when an operation times out. For calls that don't involve the return of data, the server will always set the DataBuf parameter to Nil. Thus you must be careful about the parameter you supply for DataBuf so that you don't lose your reference to a piece of memory inadvertently.

Call:

```
Function SyncIO(
            IOPort          : ServerIOPort;
            Command         : IOCommand;
            CmdBlk          : Pointer;
            CmdBlk_Cnt      : Long;
        Var DataBuf         : Pointer;
        Var DataBuf_Cnt     : Long;
            DataTransferCnt : Long;
            TimeOut         : Long;
        Var Status          : IOStatusBlk)
    : GeneralReturn
```

Parameters:

IOPort-This is a ServerIOPort that uniquely identifies the server to which the CloseIO is directed. The port to use here is the ServerIOPort returned by the server in the Open IO call.

Command-This is the IOCommand to perform.

CmdBlk-This is a pointer to the device specific command block. For most commands, you can create a pointer to an IOCmdBlk and just recast CmdBlk to point to it. For other commands (like IOWriteRegisters) which have an arbitrary number of command bytes, you will need to create a large enough buffer to accommodate your command bytes and set CmdBlk to point to it. You may also want to treat your command buffer as an IOCmdBlk for purposes of using the record field names to assign the command byte values. Note that the first two words of the command block are always treated as a CmdIdTag by the server and will be copied into the status block prior to return. The CmdIdTag is primarily useful in the AsyncIO interface to the server.

CmdBlk_Cnt–This is the number of command bytes pointed to by the CmdBlk parameters.

DataBuf–This is the pointer to the data. For commands that read data from a device, DataBuf should be set to Nil before the call. Upon return, DataBuf will point to a buffer holding the data read. When you are through with the buffer pointed to by DataBuf, you should use the InvalidateMemory call (see the document "Kernel Interface" in this manual) to deallocate the buffer. For commands that write data to a device, DataBuf will point to your buffer of data to output. Since the server will always set DataBuf to Nil prior to return for commands that don't return input data, you must take care here not to lose your pointer reference to the buffer used in the call. This is easily done by using a generic pointer variable for DataBuf that has also been set to point to your output buffer.

Databuf_Cnt–This is the number of bytes in the buffer pointed to by DataBuf. For read commands, this is the number of data bytes the server has returned to you. For read commands you should set DataBuf to Nil and DataBuf_Cnt to 0 prior to the call. (This eliminates the overhead of passing useless data in the message to the server and prevents any potential invalid memory references.) For write commands, this is the number of bytes in your output buffer.

DataTransferCnt–This is the number of bytes that you want to read/write from/to the device.

TimeOut–This is the number of microseconds that the server should wait for the operation to complete. A value of 0 means to wait indefinitely. A negative value means to not wait (when that makes sense) and otherwise has the same meaning as 0.

Status–This is the completion status of the operation. Included here is the DeviceStatus which is device dependent and may hold the values of device status registers when they were last obtained from the device. The current values are explicitly obtained when the IOSense command is issued.

Results:

IOSuccess–if the operation succeeds completely; otherwise returns an identifying error code. If the command fails, Status can also be checked for more detailed error information as well as an indication of how much of the command succeeded.

4 Name Server

4.1 Introduction

The Name Server provides processes with the ability to establish communications using textual names. In general, a process registers a Name/Port pair with the NameServer. Other processes can then ask the Name Server if it has a registration for a specific name.

If a name is not registered with the local Name Server, that Name Server will broadcast a request on the network. If the name is registered with some other Name Server on the network, that remote Name Server will reply with a Port.

At the current time the Name Server is a portion of the Message Server Process. The Message Server is responsible for extending InterProcessCommunication (IPC) over the network. The fact that the Name Server and the Message Server are the same process is transparent to a client of the Name Server.

Communication with the Name Server is through a Matchmaker-generated interface. The port that is used for communications with the Name Server is NameServerPort defined in PascalInit.pas.

4.2 Functions

The following functions are from MsgNUser.pas.

CheckIn

CheckIn is used to register a Name/Port pair with the local Name Server.

Call:

```
Function CheckIn(
            ServPort  : Port;
            PortsName : string;
            Signature : Port;
            PortsID   : Port)
        : GeneralReturn
```

Parameters:

ServPort–The port that is used to communicate with the Name Server. It is NameServerPort from PascalInit.

PortsName–The name to be registered.

Signature–This is not currently implemented. Use NullPort for this parameter.

PortsID–The port that is to be associated with PortsName. Clients of the Name Server who request a connection to PortsName will be given Send rights to PortsID.

Results:

Success–Name was registered correctly.

LookUp

LookUp is used to obtain a port that is associated with a name.

Call:

```
Function LookUp(
          ServPort  :  Port;
          PortsName :  string;
        var PortsID   :  Port )
      : GeneralReturn
```

Parameters:

ServPort–The port that is used to communicate with the Name Server. It is NameServerPort from PascalInit.

PortsName–The name to look for.

PortsID–The port that is to be associated with PortsName.The caller of LookUp will be given send rights to PortsID.

–The port PortsName was found and PortsID contains a port that can be used to communicate with it.

–The port PortsName was not found. PortsID is not valid.

CheckOut

CheckOut is used to remove a Name/Port registration from the Name Server.

Call:

```
Function CheckOut(
          ServPort  :  Port;
          PortsName :  string;
          Signature :  Port )
      : GeneralReturn
```

Parameters:

ServPort–The port that is used to communicate with the Name Server. It is NameServerPort from PascalInit.

PortsName–The name to be removed..
P3 = Signature

–This is not currently implemented. Use NullPort for this parameter.

–Name was removed.

–The client tried to remove a name that it did not register.

MsgPortStatus

MsgPortStatus returns Message Server internal information about a Port. This function is a system function and is not for users.

Call:

```
Function CheckIn(
        ServPort  :  Port;
        PortsID  :  Port;
      var GlobalPort : long;
      var Owner      : long;
      var SrcID      : long;
      var SeqNum     : long;
      var NetWaiting : boolean;
      var NumQueued  : integer;
      var Blocked    : boolean;
      var Locked     : boolean;
      var RecvQueue  : integer;
      var DataOffset : long;
      var InSrcID    : long;
      var InSeqNum   : long )
    : GeneralReturn
```

5 The Network Server

5.1 Introduction

The Network Server (or NetServer) provides access to the Ethernet for all processes that are running on a given PERQ. All access to the Ethernet is through the Network Server. The CMU 3MHz NetServer provides access to both the 3MHz and 10MHz Ethernets. On the 10MHz Ethernet it functions by encapsulating 3MHz Ethernet packets within 10 MHz Ethernet packets.

Note that this 3MHz network server is unique to CMU. At other sites a 10MHz NetServer is used by PERQs communicating solely on a 10 MHz Ethernet. See the PERQ Systems document, *The Network Server*, for more details.

User access to the Network Server is through a Matchmaker-generated message interface. The Network Server user sees a set of procedure calls. These calls are translated by Matchmaker-generated code into messages that are then sent to the Network Server. The Network Server will act on the request and send a reply to the client. The reply messages are translated by Matchmaker-generated code into a format that is suitable for the user. (Because the netserver used to be called the EtherServer, the matchmaker interface to is is called EtherUser.)

When a user process wishes to receive packets from the Ethernet, it informs the Network Server. The user provides the Network Server with the type of Ethernet packets that it wishes to receive. It also provides send rights to a port. When an Ethernet packet is received by the Network Server the Server checks to see if any of its users are interested in packets of that type. If so it will send the received packet to those users in an IPC message. To get the message cotaining a packet, the user does an IPC receive on the port that it previously provided to the Network Server to get the message that contains the packet. The user then uses Matchmaker-generated code to obtain the actual Ethernet packet from this IPC message.

5.2 overview

In order to provide both synchronous and asynchronous communication with the NetServer, the functions provided by this module are implmented with three subroutine call each. The Send and Parse calls are to be used (along with a user written Receive call) for asynchronous communication. The Wait calls are to be used for synchronous communication. The three types of routines have following implementation:

Send<X>	This routine sends off a request message to the NetServer. It usually takes a reply port as well as parameters to the function, and returns the result of the send call.
Parse<X>	This routine parses a request response from the NetServer. It usually takes a message pointer as a parameter and returns the function results as its result, or in var parameters, depending on the number of output variables.
Wait<X>	This routine combines the above two routines for synchronous server communication. It will raise an exception if something goes wrong. The

exceptions are defined below. It should be pretty obvious what their arguments refer to.

All the routines implicitly take the NetServer port as a parameter. This port is hidden withing the EtherUser module and is initialised when the routine InitEther is called.

The netserver supports two types of packet format: raw 3Megabit ether packets, and pup packets. Thus there are parallel sets of calls depending on which packet format is desired.

5.3 Initializing the interface

The procedure InitEther must be called before other routines can be executed.

Example:

```
InitEther (NullPort, 0);
```

5.4 Getting the Ethernet Address of the Machine

The Network Server user must place the Ethernet address of the current machine into the Source field of the Ethernet packet header. The Network Server provides facilities to get the Ethernet address from the Ethernet hardware. The address that is returned is in a form that can be placed directly into an address field of an Ethernet header. The CMU MHz network server always returns a 3MHz address even thought the machine may be on the 10Mhz ethernet.

Example:

```
VAR
  host    : integer;
BEGIN
  host := WaitEtherAddress;
  writeln ('My host = ', host:1);
```

5.5 Sending an Ethernet Packet

To send a packet on the Ethernet, the user must set the Ethernet packet header to contain the destination address, the correct source address and the packet type fields of the header. The packet can be sent using SendEtherPacket.

5.6 Connecting to an Ethernet Type

To receive packets from the network, the user process must tell the Network Server that it wishes to receive packets of a specific Ethernet type on a user-provided port. The Ethernet type acts as a filter within the network server. It is possible to receive packets of more than one Ethernet type on a single port by making multiple calls to WaitEtherFilter with different filters but with the same port.

Example:

```
VAR
    filter  : integer;   { Packet type }
    retval  : GeneralReturn;
    ok      : boolean;
    myport  : port;

BEGIN
  retval := AllocatePort (KernelPort, myport, DefaultBacklog);
  Write ('Filter number: ');
  Readln (filter);
  ok := WaitEtherFilter (myPort, filter);
```

5.7 Type Definitions

The following definitions are from EtherTypes.pas.

```
CONST       { return codes from PerqE3, PerqE10, PorkE3 }
 IOEIOC = #0; { successful }
 IOETIM = #1; { no packet is being returned from Ether3Receive }
 IOEPTL = #2; { a packet > WdCnt has been received }
 IOERNT = #3; { a packet < WordSize(EtherHeader) has been received }
 IOEBSE = #4; { WdCnt > E3BufferSize }
 IOEFUZ = #5; { packet length not integral number of words }

TYPE
 EtherHeader      = PACKED RECORD
            Src     : 0..255;
            Dst     : 0..255;
            Typ     : INTEGER;
           END;

CONST
 MinEtherWords     = WordSize(EtherHeader);
 MaxEtherWords     = 748; { 750 words max data size on 10mhz net, but 2
                words are needed for the 3hmz encapsulation
                header on the 10mhz net. }
               { 560 is the vax 3mhz limit (ENETPACKETSIZE)}

TYPE
 pEtherPacket     = ↑EtherPacket;
 EtherPacket     = RECORD
           CASE Integer OF
           0:(
            Header    : EtherHeader;
            DataWords  : ARRAY[0 .. MaxEtherWords -
                      WordSize(EtherHeader) - 1
                    ] OF INTEGER );
           1:(
            Words      : ARRAY[0 .. MaxEtherWords-1] OF INTEGER );
           2: (
            LongHeader : EtherHeader;
            LongWords  : ARRAY[0 .. (MaxEtherWords -
                           WordSize(EtherHeader))
                           div 2 -  1] OF LONG );
           3 : (
            ConfigHdr  : EtherHeader;
            SkipC      : integer;
            ConfigBytes : PACKED ARRAY
                         [0 .. 2 *(MaxEtherWords -
                           WordSize(EtherHeader) -  1)]
                       OF Bit8)
           END;

CONST
 EtherTypEchoMe     =  #700;
 EtherTypIAmAnEcho  =  #701;
 EtherTyp3ConfigTest =  #220;
 EtherTypPup        = #1000;
 EtherTypConfigTest  =  #220;
  FwdCode   = 2;
  ReplyCode = 1;

TYPE
 PupLLong       = RECORD
           CASE INTEGER OF
           1 : ( Lng : Long );
           2 : ( Low : Integer;
              Hgh : Integer )
           END;

TYPE
 PupHLong       = RECORD
           CASE INTEGER OF
           1 : ( Lng : Long );
           2 : ( Hgh : Integer;
              Low : Integer );
           END;


TYPE
 PupPort       = PACKED RECORD
           CASE integer OF
           0 : (
            Host: 0.. 255;
            Net : 0.. 255;
            Soc : PupHLong );
           1 : (
            All : PACKED ARRAY [0..5] OF Char )
           END;

TYPE
 PupHeader        = PACKED RECORD
            Len      : INTEGER;
            Typ      : 0.. 255;
            TC       : 0.. 255;
            Id       : PupHLong;
            Dst      : PupPort;
            Src      : PupPort
           END;

TYPE
 pPupData         = ↑PupData;
 PupData          = RECORD
           CASE INTEGER OF
           0 : (Chars  : PACKED ARRAY[ 0..1043] OF CHAR);
           1 : (Bytes  : PACKED ARRAY[ 0..1043] OF 0..255);
           2 : (Words  : PACKED ARRAY[ 0.. 521] OF INTEGER);
           3 : (HLongs : PACKED ARRAY[ 0.. 260] OF PupHLong);
           4 : (Ports  : PACKED ARRAY[ 0.. 260] OF PupPort);
           END;

CONST
 MinPupWords      = WordSize(PupHeader)+1;
 MaxPupWords      = 533;

TYPE
 pPupPacket       = ↑PupPacket;
 PupPacket        = RECORD
           CASE INTEGER OF
           0 : ( Header    : PupHeader;
                 Data      : PupData );
           1 : ( ChkSums   : ARRAY [0..532] OF Integer )
           END;

CONST
 PupCMUNet        = #52;

CONST
 PupError       =  #4;

 PupEFTPData      =  #30;
 PupEFTPAck       =  #31;
 PupEFTPEnd        = #32;
 PupEFTPBort       = #33;

CONST
 PupNameSocket     =  #4*#200000;  { word-swapped longs }
 PupEFTPSocket     = #20*#200000;

TYPE
 MsgPacket        = RECORD
              Head          : Msg;
              TPacket        : TypeType;
             END;

 MsgEtherPacket    = RECORD
              Head          : Msg;
              TPacket        : TypeType;
              Packet        : EtherPacket;
             END;

 MsgPupPacket      = RECORD
              Head          : Msg;
              TPacket        : TypeType;
              Packet        : PupPacket;
             END;

CONST
 PacketBytes      = 2 * WordSize(MsgPacket);
 EtherPacketBytes   = 2 * WordSize(MsgEtherPacket);
 PupPacketBytes    = 2 * WordSize(MsgPupPacket);

CONST
 EtherIDBase       = 800;

 IDGetEtherAddress  = EtherIDBase + 1;
 IDRGetEtherAddress  = EtherIDBase + 101;

 IDSetEtherFilter   = EtherIDBase + 2;
 IDRSetEtherFilter   = EtherIDBase + 102;

 IDClrEtherFilter   = EtherIDBase + 3;
 IDRClrEtherFilter   = EtherIDBase + 103;

 IDSendEtherPacket  = EtherIDBase + 4;
 IDRecvEtherPacket  = EtherIDBase + 104;

 IDSetPupFilter    = EtherIDBase + 5;
 IDRSetPupFilter    = EtherIDBase + 105;
 IDClrPupFilter     = EtherIDBase + 6;
 IDRClrPupFilter    = EtherIDBase + 106;

 IDSendPupPacket    = EtherIDBase + 7;
 IDRecvPupPacket    = EtherIDBase + 107;

TYPE
 pMsgGetEtherAddress = ↑MsgGetEtherAddress;
 MsgGetEtherAddress  = RECORD
             Head      : Msg
            END;

 pMsgRGetEtherAddress= ↑MsgRGetEtherAddress;
 MsgRGetEtherAddress = RECORD
             Head      : Msg;
             TEtherAddress : TypeType;
             EtherAddress  : INTEGER
            END;

 pMsgSetEtherFilter  = ↑MsgSetEtherFilter;
 MsgSetEtherFilter   = RECORD
             Head      : Msg;
             TTyp      : TypeType;
             Typ       : INTEGER;
             TProcessPort: TypeType;
             ProcessPort : Port
            END;
 pMsgRSetEtherFilter = ↑MsgRSetEtherFilter;
 MsgRSetEtherFilter  = RECORD
             Head      : Msg;
             TTyp      : TypeType;
             Typ       : INTEGER;
             TAnswer   : TypeType;
             Answer    : BOOLEAN
            END;

 pMsgClrEtherFilter  = ↑MsgClrEtherFilter;
 MsgClrEtherFilter   = RECORD
             Head      : Msg;
             TTyp      : TypeType;
             Typ       : INTEGER;
             TProcessPort: TypeType;
             ProcessPort : Port
            END;

 pMsgRClrEtherFilter = ↑MsgRClrEtherFilter;
```

```
MsgRClrEtherFilter  = RECORD
            Head      : Msg;
            TTyp      : TypeType;
            Typ      : INTEGER;
            TAnswer    : TypeType;
            Answer    : BOOLEAN
           END;

pMsgSendEtherPacket = ↑MsgSendEtherPacket;
MsgSendEtherPacket  = MsgEtherPacket;

pMsgRecvEtherPacket = ↑MsgRecvEtherPacket;
MsgRecvEtherPacket  = MsgEtherPacket;

pMsgSetPupFilter    = ↑MsgSetPupFilter;
MsgSetPupFilter     = RECORD
            Head      : Msg;
            TSocket    : TypeType;
            Socket     : Long;
            TProcessPort: TypeType;
            ProcessPort : Port
           END;
pMsgRSetPupFilter   = ↑MsgRSetPupFilter;
MsgRSetPupFilter    = RECORD
            Head      : Msg;
            TSocket    : TypeType;
            Socket     : Long;
            TAnswer    : TypeType;
            Answer     : Boolean
           END;

pMsgClrPupFilter    = ↑MsgClrPupFilter;
MsgClrPupFilter     = RECORD
            Head      : Msg;
            TSocket    : TypeType;
            Socket     : Long;
            TProcessPort: TypeType;
            ProcessPort : Port
           END;

pMsgRClrPupFilter   = ↑MsgRClrPupFilter;
MsgRClrPupFilter    = RECORD
            Head      : Msg;
            TSocket    : TypeType;
            Socket     : Long;
            TAnswer    : TypeType;
```

```
                Answer    : Boolean
               END;

  pMsgSendPupPacket  = ↑MsgSendPupPacket;
  MsgSendPupPacket   = MsgPupPacket;

  pMsgRecvPupPacket  = ↑MsgRecvPupPacket;
  MsgRecvPupPacket   = MsgPupPacket;
```

5.8 Exported Exceptions

```
Exception ESendFailed(Why: GeneralReturn);

Exception EReceiveFailed(Why: GeneralReturn);

Exception EBadReply;
```

5.9 Common Parameters

GeneralReturn	The return code from the Send Kernel call.
Listener	The port to which to send the notification that a reply to this call has been received.
MaxWait	Maximum number of milleseconds to wait for a send to NetServer to complete. The value 0 means wait forever.
MsgP	A pointer to a specific type of message.
Option	Send option for the Send call. Specifies what to do if the server queue is full. Values are *wait,dontwait*, and *reply.*
packet	A pointer to a pup packet.
Reply	Port to which reply to this message is to be sent.
Typ	A filter specifying the types of ethernet packages that are to be listened for. The values for this parameter are exported by ethertypes.pas.
socket	The local part of a pup protocol EtherNet address. This acts as a filter for pup tranmission.

5.10 Procedures

The following procedures are found in EtherUser.pas.

"InitEther"

This procedure sets up vital state information for the interface and must be called before any of the other routines are used.

Call:

```
procedure InitEther(
            RPort  : port;
            Heap   : integer )
```

Parameters:

"Rport"–port to which replies from the Ether Server should be sent. If it is equal to *NullPort*, a new port is allocated.

"Heap"–heap to be used for dynamic storage allocation. If equal to 0, the default data heap is used.

Exceptions:

""–Exceptions raised by the 'wait' routines, should be obvious, if not check the code.

"SendEtherAddress"

Send a message to find out the local EtherNet address

Call:

```
function SendEtherAddress(
              Reply  : Port;
              MaxWait: long;
              Option : SendOption )
        : GeneralReturn
```

"ParseEtherAddress"

Parse the reply message containing the EtherNet address

Call:

```
function ParseEtherAddress(
              MsgP  :pMsgRGetEtherAddress )
       :integer
```

Returns:

EtherNet address

"WaitEtherAddress"

Synchronously get the EtherNet address

Call:

```
        function WaitEtherAddress   : integer;
```

Returns:

EtherNet address

"SendEtherFilter"

Send a message to set the ether filter

Call:

```
        function SendEtherFilter(
                    Typ    : integer;
                    Listener : Port;
                    Reply    : Port;
                    MaxWait  : long;
                    Option   : SendOption)
                : GeneralReturn
```

A ether filter specifies the type of ether packets that we are interested in listening for. After a filter with the value of Typ has been set, any incoming packets of that type cause a notification message to be sent to the Listener port.

"ParseEtherFilter"

Parse reply message containing value of ether filter.

Call:

```
        function ParseEtherFilter(
                        MsgP  : pMsgRSetEtherFilter;
                    var Typ          : integer )
            :boolean;
```

Returns:

true - if the ether filter of the value Typ has been set.

false - if the filter was not set because the maximum number of filters was already set. Currently the maximum number of filters is 4.

"WaitEtherFilter"

Synchronously set a new value for the ether filter

Call:

```
function WaitEtherFilter(
            Typ   : integer;
            Listener  : Port )
        : boolean;
```

Returns:

same as for ParseEtherFilter.

"SendEtherClear"

Send message to remove an ether filter

Call:

```
function SendEtherClear(
            Typ     : integer;
            Listener : Port;
            Reply    : Port;
            MaxWait  : long;
            Option   : SendOption )
        :GeneralReturn;
```

Removes the filter of value Typ with the associated notification port of Listener.

"ParseEtherClear"

Parse message received as a result of an EtherClear message

Call:

```
function ParseEtherClear(
             MsgP    : pMsgRClrEtherFilter;
           var Typ     : integer )
        : boolean;
```

Returns:

"same as for ParseEtherClear."

"SendEtherPacket"

Send an ethernet packet pointed to by 'Packet', of size 'PacketWords'

Call:

```
function SendEtherPacket(
            Packet     : pointer;
            PacketWords: integer;
            MaxWait   : long;
```

```
                    Option    : SendOption )
              : GeneralReturn;
```

Parameters:

"Packet"–pointer to packet to be sent

"PacketWords"–number of words in packet.

"ParseEtherPacket"

Copies an ethernet packet from the response message into the area pointed at by 'Packet', returning the number of words copied.

Call:

```
function ParseEtherPacket(
                    MsgP   : pMsgRecvEtherPacket;
                    Packet       : pointer)
            : integer;
```

Parameters:

"Packet"–Pointer to packet to be parsed.

Returns:

number of words in packet

"SendPupFilter"

Add a new socket on which to receive pup packets.

Call:

```
function SendPupFilter(
                Socket   : Long;
                Listener : Port;
                Reply    : Port;
                MaxWait  : long;
                Option   : SendOption )
          : GeneralReturn;
```

"ParsePupFilter"

Parse return message from SendPupFilter call

Call:

```
function ParsePupFilter(
                MsgP     : pmsgrsetpupfilter;
            var Socket   : Long )
        : boolean;
```

Returns:

true - if the socket has been added to the list of available sockets.

false - if the socket was already in use, or if the maximum number of sockets was already allocated. Currently, the maximum number of sockets is 4.

"WaitPupFilter"

Synchronously add a new socket on which to receive pup packets.

Call:

```
function WaitPupFilter(
              VAR Socket   : Long;
                  Listener  : Port )
          : boolean;
```

Returns:

same as for ParsePupFilter.

"SendPupClear"

Send message to deallocate a socket.

Call:

```
function SendPupClear(
                Socket    : Long;
                Listener  : Port;
                Reply     : Port;
                MaxWait   : long;
                Option    : SendOption )
          : GeneralReturn;
```

"ParsePupClear"

Parse message reply from SendPupClear message.

Call:

```
function ParsePupClear(
        MsgP   : pMsgRClrPupFilter;
      var Socket : Long )
    : boolean;
```

Returns:

true - if the socket was found and deallocated.

false - if the socket was not found.

"WaitPupClear"

Synchronous call to allocate a pup socket.

Call:

```
function WaitPupClear(
        var Socket   : Long;
          Listener  : Port )
    : boolean;
```

Returns:

same as for ParsePupClear

"SendPupPacket"

" Send a Pup Packet pointed to by 'PupPacket'. "

Call:

```
    "function SendPupPacket(
 PupPacket      : pointer;
        MaxWait       : long;
        Option        : SendOption
: GeneralReturn;" )
```

"ParsePupPacket"

Copies the Pup packet from the PacketMessage into the 'PupPacket'.

Call:

```
procedure ParsePupPacket(
        MsgP     : pMsgRecvPupPacket;
        PupPacket : pointer);
```

"FinEther"

Deallocate resources allocated for local use.

Call:

```
procedure FinEther;
```

6 Process Manager

6.1 Introduction

The Process Manager controls and monitors the activity of a process. It supplements the process control functions of the Accent kernel. Process control signals can be sent between processes to control individual or groups of processes. The user can also suspend, resume, cancel and debug processes through shell or Window Manager commands.

To use the Process Manager a process must be rgistered. If the process was created with Spawn it it automatically registered otherwise a process can be registered with the function PMRegisterProcess. A process that is not registered with the Process Manager will be harder to start up and impossible to control.

6.2 Process Trees

Processes are organized into trees. Each process in the tree has a parent and a set of children. Registering a process supplies the Process Manager with:

- the parent of the process.
- the kernel and data ports of the new process,
- a descriptive name for the process,
- the window, typescript and environment manager connections,

The parent process controls the children through Accent or the Process Manager. The parent is notified when a child dies through an emergency message that contains

1. the reason for termination - "process death" if the child gives no reason
2. run time - if available
3. load time - if available
4. elapsed time - if available

A process may reregister itself with a new parent. When a parent dies surviving children are "orphans".

6.3 Process Control Groups

Processes can be controled together through Process Control Groups. Each window can only belong to one procèss group. A group can be associated with one or more windows and a process cotnrol key typed to any window affects all of the process group. The name of the group is located in the window icon (aliases for the name are in icons of other windows in the group). If a process changes it's group, it's children become members of the new group. If the process control window is removed, the process is "out of control" and can only be controlled by shell process control commands.

6.4 Process Control Signals

The Process Manager provides 64 Process Control Signals, 7 with defined roles:

Signal	Default Action
suspend	suspend the process
resume	resume the process
status	display status in process manager window
Level1Abort	terminate process
Level2Abort	terminate process
Level3Abort	terminate process
debug	suspend process & invoke debugger
other	ignore signal

On receiving a signal a process may

1. Send - send an emergency message saying what signal has occured (resume any suspended processes).
2. Ignore - ignore signal. Level3Abort signals cannot be ingored and will terminate the process.
3. Default - perform the default action for the signal.

The action for each signal may be specified independently. There are Process Manager functions (Emergency Messages) to send a signal to an entire process group or single process in a group.

6.5 Keyboard and Window Manager control

The following commands can be typed to the shell to control processes,

```
Suspend  <string>
Resume   <string>
Debug    <string>
Kill     <string>
SetPriority <string>
```

If the string given is a group name the command will affect the group. The command will affect a process if the string given is a process number or a prefix of the process name. If the string matches more than one process name the command fails and returns NameAmbiguous.

These same commands can be selected from the Window Manager menu or with the Window Manager keyboard commands. See the *User's Guide to the Window Manager* for more information.

6.6 Debugging

The process manager can register a debug port for a process. If the process gets an uncaught exception or an addressing error, or if a signal is raised and not caught, the process is suspended and an emergency message is sent to the debug port. (This is line AccInt.SetBugPort in the kernel, but intercepts signals as well as program errors.) The port may be set up to catch all uncaught signals or only the Debug signal. A debugger can use this to keep control of a process and to intercept subsequent Debug keys (instead of starting up another debugger).

The global Environment variable "DebuggerName" is the name of the debugger to run. When the debugger is invoked, the Process Manger first tries to run the program named in "DebuggerName". If that is not found, it tries to run 'Debugger.run'. If that, in turnis missing, it runs the built-in Mace debugger. If "DebuggerName" has the value '?' the Process Manager asks the user for the name of a run file to run as the debugger.

The debugger starts up with the following environment:

InPorts↑[0]	Kernel Port of target process
InPorts↑[1]	Process Manager Port
EMPort process	Environment Manager connection for target
UserWindow, UserTypescript	Window and typescript for the debugger. Process control functions on this window affect the Debugger, not the target process (the debugger may, of course, intercept them).

6.7 Definitions

The following definitions are found in ProcMgrDefs.pas,

```
Const
    ProcMgrBase   = 3600;     { Process Manager messages and errors }
    SignalBase    = 3800;     { Signals and asynchronous returns }

    { Error returns from ProcMgr }

    UnknownProcess  = ProcMgrBase+1;
    UnknownSignal   = ProcMgrBase+2;
    UnknownAction   = ProcMgrBase+3;
    UnknownWindow   = ProcMgrBase+4;
    WindowInUse     = ProcMgrBase+5;
    NoChildren      = ProcMgrBase+6;
    UnknownPort     = ProcMgrBase+7; { NullPort given to ProcMgr }
    NameAmbiguous   = ProcMgrBase+8;
    ProcessDisowned = ProcMgrBase+9;

    { Signals }

    MinSignal     = SignalBase;
    MaxSignal     = SignalBase+63;

    { signal numbers 1..32 are reserved for Spoonix! }

    SigSuspend    = MinSignal+33;
    SigResume     = MinSignal+34;
    SigStatus      = MinSignal+35;
    SigDebug       = MinSignal+36;
    SigLevel1Abort = MinSignal+37;
    SigLevel2Abort = MinSignal+38;
    SigLevel3Abort = MinSignal+39;

type
    SignalName = integer;


    { Actions }
type
    SignalAction = (SigDefault,
                    SigIgnore,
                    SigSend);


    { Statistics returned to caller }

    StatRecord = record
        RunTime:     long;  { microseconds }
        LoadTime:    long;  { microseconds }
        ElapsedTime: long;  { ticks }
        KernelPort:  long;  { number, not port }
        Priority:    integer;
        QueueID:     integer;
        ProcName:    string; { Process name }
        IconName:    ProgStr;{ name in Icon - group name }
        State:       ProcState;
      end;

    StatArray = array[0..0] of StatRecord;
    StatList  = ↑StatArray;
```

6.8 Functions

The following functions are found in ProcMgrUser.pas.

PMRegisterProcess

Register a process with the Process Manager

Call:

```
                    function PMRegisterProcess(
                                 ServPort : Port;
                                 HisKPort : Port;
                                 HisDPort : Port;
                    ProgName : string;
                    HisWindow : Window;
                    HisTypescript: Typescript;
                    EMConn   : port;
                    Parent   : port)
                : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

HisKPort–The kernel port of the process to be registered.

HisDPort–The data port of the process to be registered.

ProgName–The name of the process, usually the .RUN file name.
Spawn registers the parameter that it calls ProcName as the ProgName.

HisWindow–The window associated with the process. NullPort if none.

HisTypescript–The typescript associated with the process.

EMConn–The port associated with a new connection to the Environment manager for this process. NullPort if none.

Parent–The kernel port of the Parent Process.

The supplied data is entered into the Process Manager Data Base and the icon name for the process's window is set to be ProgName.

PMSetSignal

Set the signal actions for a process

Call:

```
function PMSetSignal(
            ServPort : port;
            ProcPort : port;
            Signal   : SignalName;
            Action   : SignalAction)
        : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcPort–The Kernel port of the process to affect.

Signal–Signal to change action for.

Action–New Action to set for signal. Actions are:

SigSend - send a SignalMessage to the process's SignalPort with the signal as the reason.

SigIgnore - completely ignore the signal.

SigDefault - take the default action for the signal.

Results:

Success

UnknownProcess

UnknownSignal

UnknownAction

PMSetSignalPort

Sets the signal port (port to receive signal message for SigSend) for a process.

Call:

```
function PMSetSignalPort(
                ServPort  : port;
                ProcPort  : port;
                SignalPort : port )
            : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcPort–The Kernel port of the process to affect.

SignalPort–The port to receive SignalMsg

PMSetDebugPort

Sets up a port to intercept errors and default signals for a process. If the debug port exists, the default action for the debug signal (or all signals) will be to suspend the process and send a message to the debug port. In addition, the debug port will receive all Debug error messages for the process (from the kernel).

Call:

```
function PMSetDebugPort(
                ServPort  : Port;
                ProcPort  : Port;
                DebugPort : Port;
                DebugSignalOnly : Boolean)
        : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcPort–The Kernel port of the process to affect.

DebugPort–Port to receive emergency message when something happens to the process.

DebugSignalOnly–If TRUE, only intercept uncaught DEBUG signals. If FALSE, intercept ALL uncaught signals.

Results:

Success

UnknownProcess

The 'SetDebugPort' call in Accent only intercepts exceptions and memory faults. This call intercepts Signals as well.

PMSaveLoadTime

Saves the load time for a process so that it can be printed later.

Call:

```
function PMSaveLoadTime(
                ServPort  : Port;
                ProcPort  : Port;
                LoadTime  : long)
            : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcPort–Kernel port of process.

LoadTime–Time to load process (microseconds)

Results:

Success

UnknownProcess

PMGetWaitID

Get the 'Wait ID' of a child process. The Wait ID is a 32-bit number that is returned when the child process dies. Its only use is to identify the dead process (the dead process' KernelPort is no longer valid, since it was deallocated when the process died).

Call:

```
function PMGetWaitID(
            ServPort  : Port;
            ProcPort  : Port;
        var WaitID    : long)
    : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcPort–Kernel port of process.

WaitID–returns the Wait ID for the child process.

Results:

Success

UnknownProcess

The 'processdeath' message is returned to the parent's DataPort, whether or not it has asked for the child's WaitID.

PMGetTimes

Returns the run and elapsed time for a process.

Call:

```
function PMGetTimes(
            ServPort     : Port;
            ProcPort     : Port;
        var LoadTime     : long;
        var RunTime      : long;
        var ElapsedTime  : long)
    : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcPort–Kernel port of process.

LoadTime–Returns the load time for the process (microseconds).

RunTime–Returns the run time for the process (microseconds).

ElapsedTime–Returns the elapsed time for the process (1/60 second).

Results:

Success

UnknownProcess

PMGetProcPorts

Get the registered ports of a process.

Call:

```
function PMGetProcPorts(
                  ServPort  : Port;  {get ports for process being
                  ProcPort  : Port;   debugged}
                var hisWindow : window;
                var histypescript : typescript;
                var hisEMConn : Port)
    : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcPort–Kernel port of process.

hisWindow–returns window

hisTypescript–returns typescript for process.

hisEMConn–returns Environment connection for process.

Results:

Success

UnknownProcess

PMTerminate

Terminate a process.

Call:

```
function PMTerminate(
            ServPort  : Port;
            ProcPort  : Port;
            Reason    : long)
      : GeneralReturn
```

Parameters:

ServPort–Port to Process (exported by PascalInit as PMPort).

ProcPort–Kernel port of process.

Reason–Reason for termination.

Results:

Success

UnknownProcess

PMDebugProcess

Invoke a debugger on a process.

Call:

```
function PMDebugProcess(
            ServPort  : Port;
            ProcPort  : Port;
            Reason    : long)
      : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcPort–Kernel port of process.

Reason–Reason for debugging process.

Results:

Success

UnknownProcess

Failure–couldn't invoke debugger

PMAddCtlWindow

Add a new window to the set of controlling windows for a process group.

Call:

```
function PMAddCtlWindow(
          ServPort : Port;
          CtlWindow : Window;
          NewCtlWindow : Window)
    : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

CtlWindow–an existing window for a control group.

NewCtlWindow–a new window to add as a control window.

Results:

Success

UnknownWindow–CtlWindow is not a control window for a process group.

UnknownPort–NewCtlWindow is NullPort.

PMRemoveCtlWindow

Removes a window to the set of controlling windows for a process group.

Call:

```
function PMRemoveCtlWindow(
          ServPort : Port;
          CtlWindow : Window)
    : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

CtlWindow–an existing window for a control group.

Results:

Success

UnknownWindow–CtlWindow is not a control window for a process group.

PMChangeGroup

Changes a process (and its descendants) to a new process group. If there is already a process in the process group, it must be the parent of the affected process.

Call:

```
function PMChangeGroup(
            ServPort  : Port;
            ProcPort  : Port;
            NewWindow  : Window)
    : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcPort–Kernel port of process.

Window–Window to use for new group.

Results:

Success

UnknownProcess

UnknownPort–if window is NullPort.

WindowInUse–window already controls a process group.

PMGroupSignal

Removes a window to the set of controlling windows for a process group.

Call:

```
function PMGroupSignal(
            ServPort  : Port;
            CtlWindow : Window;
            Signal    : SignalName)
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

CtlWindow–One of the windows controlling a process group.

Signal–Signal to send.

This is an Emergency message to the process manager.

PMProcessSignal

Sends a signal to a single process.

Call:

```
function PMProcessSignal(
          ServPort : Port;
          ProcPort : Port;
          Signal   : SignalName)
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcPort–Kernel port for a process to signal.

Signal–Signal to send.

This is an Emergency message to the process manager.

PMSuspend

Suspend a named (or numbered) process.

Call:

```
function PMSuspend(
          ServPort : Port;
          ProcID   : string)
: GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcID–Name of group, prefix of processname or kernel port number of process as returned by PMGetStatus.

Results:

Success

UnknownProcess

NameAmbiguous

If the name designates a process group, it sends the Suspend signal to the process group. If the name designates a single process, it suspends the process. If the process has a DebugPort set, it also send an M_DebugMsg message to the debug port, with SigSuspend as the reason (unless the debug port is set to DebugSignalOnly).

PMResume

Resume a named (or numbered) process.

Call:

```
function PMSuspend(
        ServPort  : Port;
        ProcID    : string)
    : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcID–Name of group, prefix of processname or kernel port number of process as returned by PMGetStatus.

Results:

Success

UnknownProcess

NameAmbiguous

If the name designates a process group, it sends the Suspend signal to the process group. If the name designates a single process, it suspends the process. If the process has a DebugPort set, it also send an M_DebugMsg message to the debug port, with SigSuspend as the reason (unless the debug port is set to DebugSignalOnly).

PMDebug

Invokes the debugger on a named (or numbered) process.

Call:

```
function PMDebug(
        ServPort  : Port;
        ProcID    : string)
    : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcID–Name of group, prefix of processname or kernel port number of process as returned by PMGetStatus.

Results:

Success

UnknownProcess

NameAmbiguous

If the name designates a process group, it sends the Suspend signal to the process group. If the name designates *a single process, it suspends the process. If the process has a DebugPort set, it also send an M_DebugMsg message to the debug port, with SigDebug as the reason.*

PMKill

kill a named (or numbered) process.

Call:

```
function PMKill(
        ServPort  : Port;
        ProcID    : string)
    : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcID–Name of group, prefix of processname or kernel port number of process as returned by PMGetStatus.

Results:

Success

UnknownProcess

NameAmbiguous

If the name designates a process group, it sends the Suspend signal to the process group. If the name designates a single process, and the process does not have a DebugPort set, it terminates the process with reason= SigLevel1Abort. If the process has a DebugPort set, it instead sends a M_DebugMsg message to the debug port, with SigLevel1Abort as the reason (unless the debug port is set to DebugSignalOnly).

PMSetPriority

Sets the priority of a named (or numbered) process. or of all of the processes in a process group.

Call:

```
function PMSetPriority(
        ServPort  : Port;
        ProcID    : string;
        priority  : integer)
    : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcID–Name of group, prefix of processname or kernel port number of process as returned by PMGetStatus.

Priority–Desired run priority for process:
0: lowest priority
15: highest priority

Results:

Success

UnknownProcess

NameAmbiguous

BadPriority

Changes the priority of the process or of all processes in the process group.

PMBroadcast

Print a message in the Process Manager Window. This is used to display 'system' messages.

Call:

```
function PMBroadcast(
         ServPort  : Port;
         s         : string)
    : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

s–String to display.

Results:

success

PMGetStatus

Returns status for one or more processes.

Call:

```
function PMGetStatus(
         ServPort  : Port;
         ProcID    : string;
      var Stats    : StatList;
      var Stats_Cnt : long)
    : GeneralReturn
```

Parameters:

ServPort–Port to Process Manager (exported by PascalInit as PMPort).

ProcID–Can be:

Process group name -
 returns status for all processes in group.
Pattern to match (as in file name patterns) -
 returns status for all matching process names
Process number for process -
 returns status for that process.
Null String;
 returns status for all registered processes.

Stats–Returns a pointer to the array of status records, one for each process whose name matches ProcID.

Stats_Cnt–Returns number of records in Stats

Results:

Success

UnknownProcess–name didn't match any process

Asynchronous (Emergency) Messages

Message sent to DebugPort to report an action on another process. (This is the same as the kernel's M DebugMsg.)

Parameters:

KPort–Kernel port of process affected.

Arg1–Always 0 (present for historical reasons)

Arg2–Reason for DebugMessage:

if the process was halted by an error, this is
 the GeneralReturn value describing the error
 (MemFault, UncaughtException, ...)
if the process has a Process Manager Debug Port set
 (by PMSetDebugPort), and a signal was raised on
 process and not sent or ignored, this is the name
 of the signal that was raised.

```
type DebugMessage = record
    Head        : Msg;   {Accent message header}
    tKPort     : TypeType; { (TypePT, 32) }
    KPort      : Port    { Kernel port of process affected }

    tArg1      : TypeType; { (TypeInt32, 32) }
    Arg1       : Long     { What happened, field 1: always 0! }

    tArg2      : TypeType; { (TypeInt32, 32) }
    Arg2       : Long;    { What happened, field 2:
                      GeneralReturn value for
                      error or signal. }
end;
```

Asynchronous (Emergency) Messages

Message sent on process termination

Parameters:

WaitID–Wait ID of process that died, as returned from PMGetWaitID.

Reason–Reason for process death:
if killed by terminate (in AccInt),
PROCESSDEATH

if killed by PMTerminate (in ProcMgr),
the Reason given to PMTerminate.

if the process re-registered with a different parent,
ProcessDisowned.

if the process was killed by a SigLevel[1,2,3] Abort,
the signal value.

LoadTime–CPU time to load process (microseconds)

RunTime–CPU time to run process (microseconds)

ElapsedTime–Total time elapsed from PMRegisterProcess until
termination (60Hz clock ticks).

```
const ProcessDeathMsgID= 3800;

type ProcessDeathMsg = record
    Head        : Msg;  {Accent message header}
    tWaitID     : TypeType;  { (typeInt32, 32) }
    WaitID      : long;    { Wait ID of process,
                    returned from PMGetWaitID }
    tReason     : TypeType;  { (TypeInt32, 32) }
    Reason      : long;    { reason for process death }

    tLoadTime    : TypeType;  { (TypeInt32, 32) }
    LoadTime     : long;    { process Load time
                    (microseconds) }

    tRunTime     : TypeType;  { (TypeInt32, 32) }
    RunTime      : long;    { process Run time
                    (microseconds) }

    tElapsedTime  : TypeType;  { (TypeInt32, 32) }
    ElapsedTime   : long;    { Elapsed time
                    (ticks) }
end;
```

Asynchronous (Emergency) Messages

Message sent for signal to a process: this is sent to a process's SignalPort, or to its DATAPORT if the SignalPort has not been set or is NullPort.

Parameters:

CtlWindow–Window that the signal was sent from. If the signal was sent only to one process, it is the window that heads the control group for the process.

Signal–signal sent to process.

```
const SignalMsgID = 3801;

type SignalMsg = record
    Head        : Msg;    {Accent message header}

    tCtlWindow   : TypeType; { (TypePt, 32) }
    CtlWindow    : window;  { Window that signal was
                    received on }
    tSignal      : TypeType; { (TypeInt16, 16) }
    Signal       : SignalName;  { signal received }
end;
```

7 Sesame: Interface to Prelimary Sesame File System

Sesame provides routines to read, write and get information about files, given an absolute pathname for the file. The port to this server (the ServPort parameter) is exported as SesPort from PascalInit. See the section on Sesamoid in the *Sesame: The Spice File System* Manual for more information.

7.1 Type Definitions

The following definitions are from SesameUser.pas.

```
const
  Path_Name_Size    = 255; { Number of characters in a Path_Name }
  Entry_Name_Size   = 80;  { Number of characters in an Entry_Name }
type
  APath_Name       = string[Path_Name_Size];  { An abs. pathname }
  Wild_APath_Name   = string[Path_Name_Size];  { A wild abs. pathname }
  Entry_Name       = string[Entry_Name_Size]; { A pathname component }


{ Name_Flags: Flags giving desired treatment of names in Name Server calls.
{     Note that specific flag values may be illegal for certain calls, and
{     must be zero.


const
  NFlag_Deleted   = #000001; { Allow deleted names }
  NFlag_NoNormal  = #000002; { Disallow normal (not deleted) names }
  NFlag_RESERVED  = #177774; { These bits reserved for expansion }
type
  Name_Flags     = 0 .. #3;


{ Name_Status: Flags useful for determining the disposition of a name in
{     the name data base.
const
  NStat_Deleted   = #000001; { Set if name is deleted }
  NStat_High      = #000002; { Set if name is highest undeleted version }
  NStat_Low       = #000004; { Set if name is lowest undeleted version }
  NStat_RESERVED  = #177770; { These bits reserved for expansion }
type
  Name_Status    = 0 .. #7;


{ Entry_Type: The kinds of objects which can be in the name data base.
```

```
const
  Entry_All          = 0;   { Special value referencing all entry types }
  Entry_File         = 1;   { Entry_Data is a File_ID }
  Entry_Directory    = 2;   { Name refers to another level of the
                      name hierarchy.  Entry_Data is empty. }
  Entry_Port         = 3;   { Entry_Data is a port }
type
  Entry_RESERVED     = 4 .. #377; { These values reserved for expansion }
  Entry_UserDefined  = #400 .. #77777; { Values available to the user }
type
  Entry_Type         = 0 .. #77777;


{ Entry_Data:  The variant data record dependent upon the Entry_Type value.

type
  Entry_Data = record case Entry_Type of
    Entry_File    : ( {EDFileID : File_ID  -  Not Yet Implemented} );
    Entry_Directory : ( );
    Entry_Port    : ( EDPort : Port );
    #400          : ( EDBytes : packed array [0..255] of bit8 );
    #401          : ( EDWords : array [0..127] of integer );
    #402          : ( EDLongs : array [0..63] of long );
    #403          : ( EDString: string[255]);
    end;
{ Entry_List_Record:  ScanNames returns array of Entry_List_Record.


  Entry_List_Record   = record
    EntryName     : Entry_Name;
    EntryVersion  : long;
    EntryType     : Entry_Type;
    NameStatus    : Name_Status;
    end;

  Entry_List_Array    = array [0..0] of Entry_List_Record;  { hack }
  Entry_List          = ↑ Entry_List_Array;


{ Valid_Name_Chars:  Those 7-bit ascii characters which can occur in an
{     entry name without being quoted.  Note that to match uppercase,
{     uppercase letters also have to be quoted.

const
  Valid_Name_Chars =        { Put in a string since can't put in a set }
    '$-.+0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz';
```

```
{ Separator characters for parts of Path_Name syntax.

const
   Dir_Separator   = '/';  { Directory separator }
   Ver_Separator   = '#';  { Version number separator }
   Quote_Char      = '\';  { Quote character for "funny" characters }

   Up_one_Directory = '../';
   This_Directory   = './';

{ Group_ID:  An identifier for a User Group

type
   Group_ID   = long;


{ File_Data:  A pointer to a file mapped into memory


type
   File_Data  = pointer;          { A pointer to data for file calls }


{ Print_Name:  A short string name in the file header.


const
   Print_Name_Size = 80;     { Number of characters in a file Print_Name }
type
   Print_Name  = string[Print_Name_Size];    { A print name string }


{ Data_Format:  A user-defined longword for storing data format information.
{     System-defined values are given below.  User-defined values must
{     have a non-zero high word (#200000 or greater).  These values are
{     intended to be used by programs which need to know about data formats
{     in order to convert from one format to another (such as FTP).

const
   DForm_Unspecified  = 0;        { Unspecified data format } .

   DForm_8_Bit      = 8;       { 8 bit binary data }
   DForm_16_Bit     = 16;      { 16 bit binary data }
   DForm_32_Bit     = 32;      { 32 bit binary data }
   DForm_36_Bit     = 36;      { 36 bit binary data }
```

```
  DForm_CRLF_Text    = #413;      { CRLF delimited text }
  DForm_LF_Text     = #410;      { LF delimited text }

  DForm_Press       = #1000;     { Press file format data }
type
  Data_Format = long;


{ File_Header: How the user perceives the file header information.  It is
{      actually generated by the GetFileHeader style calls from the real
{      header.


type
  File_Header = packed record
    FileSize      : long;       { Size of file in bytes }
    DataFormat    : long;       { Advisory data format }
    PrintName     : Print_Name;  { The file print name }
    Author        : Group_ID;    { Who wrote the file }
    CreationDate  : Internal_Time;{ When it was written }
    AccessID      : Group_ID;    { Who last accessed the file }
    AccessDate    : Internal_Time;{ When it was last accessed }
    FHdr_RESERVED  : array [56..64] of integer;   { Pad for expansion }
    end;

{ Error return values

const
  Sesame_Error_Base   = 1200;

  NameNotFound       = Sesame_Error_Base + 1;
  DirectoryNotFound  = Sesame_Error_Base + 2;
  DirectoryNotEmpty  = Sesame_Error_Base + 3;
  BadName          = Sesame_Error_Base + 4;
  InvalidVersion     = Sesame_Error_Base + 5;
  InvalidDirectoryVersion
                 = Sesame_Error_Base + 6;
  BadWildName       = Sesame_Error_Base + 7;
  NotAFile          = Sesame_Error_Base + 8;
  NoAccess          = Sesame_Error_Base + 9;
  NotSamePartition    = Sesame_Error_Base + 10;
  ImproperEntryType   = Sesame_Error_Base + 11;
  NotADirectory      = Sesame_Error_Base + 12;
```

7.2 Procedures and Functions

The following procedures and functions are found in SesameUser.pas.

Several of the calls are provided with two or more forms, with one being a subset of the other. Where this is the case, the subset form will have a name prefix of Sub, whereas the full form of the call will have a name prefix of Ses.

InitSesame

Initialize the local copy of this module.

Call:

```
procedure InitSesame(RPort : port);
```

SubReadFile

Simplest way to read a file.

Call:

```
Function SubReadFile(
          ServPort : port;
          APathName : APath_Name;
        var Data : File_Data;
        var Data_Cnt  : long)
              : GeneralReturn
```

Parameters:

ServPort–port to the Name Server

apathname–the absolute pathname to read the data from

data pointer–a pointer to the data read in memory

Data_Cnt–the total number of bytes read

Results:

success–the data was successfully mapped into memory

no such name–no entry was found under apathname

access violation–the requesting client tried to read a file
for which he did not possess rights

not a file–the entry found under apathname was not a file

no such file ID–the file associated with the entered
ID could not be found

The SubReadFile request is used to map the data associated with the given filename into memory. Note that we *map the entire file in, thus there are no arguments specifying the size or position of the data to be read. We can get away with mapping the whole file since we actually will only bring in the pages which are touched, the rest being backed to secondary storage. Holes in the file will be mapped in as valid zero pages which will actually be created if written to.*

SesReadFile

Read a file with complete specification

Call:

```
Function SesReadFile(
        ServPort : port;
      var APathName : APath_Name;
      var Data : File_Data;
      var Data_Cnt  : long;
      var DataFormat : Data_Format;
      var CreationDate: Internal_Time;
      var NameStatus : Name_Status)
    : GeneralReturn
```

Parameters:

ServPort–port to the Name Server

apathname–the absolute pathname to read the data from

Data–a pointer to the data read in memory

Data_Cnt–the total number of bytes read

DataFormat–one of {unspecified, text, bit8, bit16, bit32, bit36, press, ...}

CreationDate–the date and time the file was written

NameStatus–low version, high version

Results:

success–the data was successfully mapped into memory

no such name–no entry was found under apathname

access violation–the requesting client tried to read a file for which he did not possess rights

not a file–the entry found under apathname was not a file

no such file ID–the file associated with the entered ID could not be found

The SesReadFile request is a long form of the SubReadFile call. It returns several parameter values which the short form does not.

SubWriteFile

Simplest way to write a file

Call:

```
Function SubWriteFile(
            ServPort : port;
        var APathName : APath_Name;
            Data : File_Data;
            Data_Cnt  : long;
            DataFormat : Data_Format;
        var CreationDate: Internal_Time)
    : GeneralReturn
```

Parameters:

ServPort–port to the Name Server

Apathname–the absolute pathname to write the data to

Data–a pointer to the data in memory

Data_Cnt–the number of bytes to be written

DateFormat–one of {unspecified, text, bit8, bit16, bit32, bit36, press, ...}

CreationDate–the date and time the file was written

Results:

success–the data was written under the name returned

access violation–the requesting client tried to write a file for which he did not possess rights

conflicting version–the explicit version number was less than or equal to that of an existing version

allignment error–the data was not on a page boundry

The SubWriteFile request is used to enter a new name into the directory structure and write a file under that name. This short form of the call picks up defaults for unspecified parameters from previous versions or directory defaults. The user must have CreateNames rights in the directory or Supercede rights on the previous version. Empty pages need no disk pages assigned to them.

SesGetFileHeader

Get file header information

Call:

```
Function SesGetFileHeader(
        ServPort : port;
        APathName : APath_Name;
      var FileHeader : File_Header)
    : GeneralReturn
```

Parameters:

ServPort–port to the Name Server

Apathname–the absolute pathname with which to find the file

FileHeader–a data structure containing the
fields of the file header.

Results:

success–the header data was successfully returned

no such name–no entry was found under apathname

access violation–the requesting client tried to read a file header
for which he did not possess rights

not a file–the entry found under apathname was not a file

no such file ID–the file associated with the entered ID
could not be found

The SesGetFileHeader request is used to return fields from the file header.

SesReadBoth

Read the contents and file header information of a file

Call:

```
Function SesReadBoth(
        ServPort : port;
      var APathName : APath_Name;
      var Data : File_Data;
      var Data_Cnt  : long;
      var FileHeader : File_Header;
      var NameStatus : Name_Status)
    : GeneralReturn
```

Parameters:

ServPort–port to the Name Server

Apathname–the absolute pathname with which to find the file

Data–a pointer to the data read in memory

Date_Cnt–the total number of bytes read

FileHeader–a data structure containing the
fields of the file header.

NameStatus–deleted/undeleted, low version, high version

Results:

success–the header data was successfully returned

"no such name"–no entry was found under apathname

"access violation"–the requesting client tried to read a file
header for which he did not possess rights

"not a file"–the entry found under apathname was not a file

"no such file ID"–the file associated with the entered ID
could not be found

The SesReadBoth request is used to return both the data from a file and fields from its header.

SubLookUpName

Lookup a file ID

Call:

```
Function SubLookUpName(
            ServPort : port;
          var APathName : APath_Name;
          var EntryType : Entry_Type;
          var EntryData : Entry_Data;
          var NameStatus : Name_Status)
        : GeneralReturn
```

Parameters:

ServPort–port to the Name Server

Apathname–the name to be looked up (may not
contain wildcard characters)

EntryType–the type value of the entry found. Some example values are *File*, *Directory*, and *Port*.

EntryData–a variant field dependent upon the *entry type*.
A File ID will be returned here for a *File* entry, whereas there will be no data returned for a *Directory* – all that this call will tell you about a directory is that it exists.

NameStatus–deleted/undeleted, low version, high version

Results:

success–the name was successfully looked up.

name not found–the specified name was not found

access violation–the client does not have sufficient rights to look up the given name

This function provides a simple way to look up a name. If any entry encountered during the parsing of apathname is of type Symbolic Link, a macro expansion is performed using the value of that entry in place of the corresponding name in apathname. If the final result is of type FID or IPC Port, the corresponding FID or IPC port is returned. If it is of type Directory no entry data is returned, but entry type specifies the fact that a lookup on a directory name was done. If apathname does not contain a version number, the most recent version is assumed.

SubTestName

See if a file exists

Call:

```
Function SubTestName(
            ServPort: port;
          var APathName : APath_Name;
          var EntryType : Entry_Type;
          var NameStatus : Name_Status)
      : GeneralReturn
```

Parameters:

ServPort–port to the Name Server

Apathname–the name to be looked up (may not contain wildcard characters)

EntryType–the type value of the entry found. Some example values are *File*, *Directory*, and *Port*.

NameStatus–deleted/undeleted, low version, high version

Results:

success–the name was successfully found

name not found–the specified name was not found

access violation–the client does not have *UseNames* rights on the name's directory

SubTestName is like SubLookupName except that it never returns the entry data associated with a name, but only the entry type. Note that this only requires UseNames rights on the parent directory, whereas SubLookupName requires Lookup rights on the name itself – a much stronger requirement. It is anticipated that this call will be used when all that is desired is to test for the existance of a name, and to determine its type.

SubEnterName

Enter a file or port name in a directory

Call:

```
Function SubEnterName(
                 ServPort : port;
               var APathName : APath_Name;
                 EntryType : Entry_Type;
                 EntryData : Entry_Data)
         : GeneralReturn
```

Parameters:

ServPort–port to the Name Server

Apathname–the name to be entered into the directory structure

EntryType–the type value of the object to be entered. Some example values are *File*, *Directory*, and *Port*.

EntryData–a variant field dependent upon the *entry type.* For instance, this field must contain a File ID for type *File*, and an IPC port for type *Port*. For type *Directory*, this field is left empty, seeing as how the user can't write any directory data directly. Use of this call with *entry type Directory* enters a new directory, thus no special call is needed for that purpose.

Results:

success–*apathname* was successfully entered *conflicting version*–the version number specified in *apathname* was less than or equal to that of an already existing version

access violation–the client does not possess *CreateNames* rights in the specified directory

invalid directory version–only one version number of a directory is allowed

The SubEnterName request is used to place a name and entry value pair in the directory structure. It requires CreateNames permission in the directory within which the name is being entered if no version of the name already exists else if a version of the name already exists, then Supersede privileges on the name are needed. If a previous version of the name already exists in the directory and no version number is specified in apathname, the name is entered with the next higher version number. If a previous version of the name already exists and a version number is specified in the name, then it must be greater than the highest version number so far. If no previous version of the name has ever existed, then version one is assigned if none is specified in apathname, otherwise the specified version is used.

SubDeleteName

remove a name from a directory

Call:

```
Function SubDeleteName(
                ServPort : port;
                APathName : APath_Name)
               : GeneralReturn
```

Parameters:

ServPort–the name to be deleted

Apathname

Results:

success–the name was successfully deleted

nonexistent name–the name specified does not exist

access violation–the user does not possess *DeleteNames* rights on the name specified

name already deleted–the given name was already deleted

directory not empty–illegal to delete a non-empty directory

The SubDeleteName request is used to delete a name from a directory. It requires DeleteNames permission in *the directory or Delete permission on the name. The name is marked as deleted, but remains in the directory and is queued for expunging by the Migration Server. If no version number is specified then the lowest version in the directory will be deleted.*

SubReName

rename a file or port

Call:

```
Function SubReName(
          ServPort : port;
          OldAPathName : APath_Name;
        var NewAPathName : APath_Name)
        : GeneralReturn
```

Parameters:

ServPort–port to the Name Server

OldAPathName–absolute pathname of the object to be renamed

NewAPathName–new name to enter the object under

Results:

success–the name was successfully changed

access violation–the client either does not have the proper rights to delete *old apathname* or to enter *new apathname*

SubRename enters the object specified by old apathname into into the global name space as new apathname and then removes the old apathname. The access control list of the object is moved with it.

SesScanNames

Returns a list of names in a directory

Call:

```
Function SesScanNames(
          ServPort : port;
          WildAPathName : Wild_APath_Name;
          NameFlags : Name_Flags;
          EntryType : Entry_Type;
        var DirectoryName : APath_Name;
        var EntryList : Entry_List;
        var EntryList_Cnt : long)
      : GeneralReturn
```

Parameters:

ServPort–the absolute pathname to be scanned
(may contain wildcard characters in the terminal component)

WildAPathName

NameFlags–inhibit/allow deleted/undeleted names

EntryType–the entry type being scanned for. The special type designator *All* may be given, in which case names of all entry types are returned.

DirectoryName–the absolute pathname of the directory in which matches occured

EntryList–the sorted entry names, types, version numbers and name status flags of all names matching *wild apathname* in the first element of *search list* where a match occured

EntryList_Cnt–the actual number of list elements returned. Will be zero if no match occured.

Results:

success–the given wild absolute pathname was successfully scanned

access violation–the client does not possess *ReadNames* rights on the directory to be scanned

illegal pathname–wildcards were found in *wild apathname* at other than the terminal component

The SesScanNames call is used to search for a given pattern in a specified directory. It will sort and return all the matches to the given pattern in the directory. Optionally, names only of a specific entry type can be scanned for. Symbolic links are not expanded, and are returned by this call. Depending on the value of name flags, only active names, deleted names or both are considered. If no version number is specified, then the highest existing version of the name is returned. If the version number is wildcarded, then all existing versions of the name are returned. The version field for directory and symbolic link entries will be returned as zero. Note that wildcarding is permitted **only** *in the terminal component of wild apathname. This call requires either ReadNames access on the directory being scanned, in which case all matches will be returned, or else Visible access on each match which is to be returned.*

8 TimeServer

8.1 Introduction

The Time Server provides all of the time facilities for Accent. It can return the time in a number of different formats.

Internally, time is kept as the number of milliseconds since midnight, November 17,1858 Greenwich Mean Time (the Smithsonian time standard). In order to store the time efficiently, it is kept as an ordered pair, the first component representing the number of weeks that have passed since the base date-time and the second component representing the number of milliseconds that have passed in that week. This format also allows efficient comparison of times. Conversions to local time are done using a time zone index and information regarding whether or not to apply daylight savings time.

The zone record, Zone_Info, allows you to specify and receive zone information. Both the zone number and application of daylight savings time may be either specified explicitly or defaulted, depending on the settings of the UseTimeZone and UseDaylight bits. The zone record is used in User_Time.

The date and time information in User_Time is broken down into fields for input and output. Both the time zone index and application of daylight savings time can be either explicitly specified or defaulted through use of the Zone_Info fields. The Weekday field is unused for input.

8.2 Definitions

The following definitions are from TimeDefs.pas.

```
 Internal_Time:   a record containing the date and time in Greenwich
     Mean Time.  This is the Smithsonian Institute's time standard.
     To optimize space usage, we store
     time as an ordered pair, the first representing the number of
     weeks which have passed since 17-Nov-1858, when the
     Smithsonian time standard began. The second represents the
     number of milliseconds which have passed in that week.

type
  Internal_Time   = record
    Weeks      : integer;  { Number of weeks since 17-Nov-1858 }
    MSecInWeek  : long;     { Number of milliseconds in that week }
    end;


 Date_Fields:  fields necessary for representing date information
     without respect to the time.  Used in by User_Time.

type
  Date_Fields     = packed record
```

```
    Year       : integer; { Such as 1982 }
    Month      : 1 .. 12; { 1 = January, 12 = December }
    Day        : 1 .. 31;
    Weekday    : 0 .. 6;  { 0 = Monday, 6 = Sunday (output only) }
    end;
```

Time_Fields: fields necessary for representing time information without respect to the date. Used in User_Time.

```
type
  Time_Fields   = packed record
    Hour        : 0 .. 24;
    Minute      : 0 .. 59;
    Second      : 0 .. 59;
    Millisecond : 0 .. 999;
    end;
```

Zone_Info: this record allows the user to specify and receive time zone information. Both the zone number and application of daylight savings time may be either specified explicitly, or defaulted, depending upon the settings of the UseTimeZone and UseDaylight bits. Used in User_Time.

```
type
  Zone_Info  = packed record
    TimeZone    : integer; { Increasing minutes west from GMT.
                             GMT = 0, EST = 5*60, CST = 6*60, ...
                             Used only if UseTimeZone is set. }
    UseTimeZone : boolean; { True when TimeZone field is valid,
                             else false when local time zone is
                             to be used. }
    Daylight    : boolean; { True if daylight savings time is to
                             be applied. Used only if
                             UseDaylight is set. }
    UseDaylight : boolean; { True if Daylight savings field is
                             valid, else false when the system
                             default for daylight savings time
                             application is to be used. }
    end;
```

User_Time: Date and Time information broken down into fields as the user would want to use it for input and output. Both the time zone index and application of daylight savings time can be

either explicitly specified, or defaulted through use of the
ZoneInfo fields. The Weekday field is unused for input.

```
type
  User_Time     = packed record
    Date      : Date_Fields;
    Time      : Time_Fields;
    Zone      : Zone_Info;
    end;
```

The following flag values may be ORed together to form TimeFormat values.

```
const
  TF_Weekday     = #000001; { If set output the day of the week
                    according to the setting of
                    TF_FullWeekday else don't output
                    the day of the week }
  TF_FullWeekday  = #000002; { If set output full text for the
                    weekday else the 3-letter
                    abbreviation (Monday/Mon) }
  TF_NoDate      = #000004; { If set do not output date and ignore
                    flags through TF_NoTime }
  TF_FullMonth   = #000010; { If set output full text for the
                    month when the month is alphabetic
                    else the 3-letter abbreviation
                    (March/Mar) }
  TF_FullYear    = #000020; { If set output the year as a 4-digit
                    number else the year is is output as
                    a 2-digit number if in the range
                    1900-1999 (1982/82) }
{ The next six settings are mutually exclusive }

  TF_Dashes     = #000000; { Output date as day-month-year
                   (22-Mar-60) }
  TF_Spaces     = #000040; { Output date as day month year
                   (22 Mar 60) }
  TF_Reversed    = #000100; { Output date as month day, year
                   (Mar 22, 60) }
  TF_Slashes    = #000140; { Output date as month/day/year
                   (03/22/60) }
            { #000200 is reserved for future expansion}
            { #000240 is reserved for future expansion}
  TF_ANSI       = #000300; { Output date according to ANSI X3.30-1971.
                    Also slightly affects time formatting.
                    (600322) }
  TF_ANSI_Ordinal = #000340; { Similar to TF_ANSI but 3-digit day-of-year
                    instead of month and day.
                    (60082) }
  TF_DateFormat  = #000340; { A mask allowing us to examine the above. }

  TF_NoTime      = #000400; { If set do not output time and ignore
                    flags through TF_NoColumns }

{ The next two settings are mutually exclusive }
  TF_NoSeconds   = #001000; { If set do not output the seconds }
  TF_Milliseconds = #002000; { If set output milliseconds as
                    hh:mm:ss.sss else omit them
                    (17:00:00.001/17:00:00) }

  TF_12_Hour     = #004000; { If set output the time in 12-hour
                    format with 'am' or 'pm' following the
                    time else output in 24- hour format.
                    Note that exact NOON outputs neither 'am'
                    nor 'pm' because 12:00am is 0000 and 12:00pm
                    is 2400. Use of TF_12_Hour with TF_ANSI or
                    TF_ANSI_Ordinal is supported but not
                    recommended for a number of reasons. If used
                    with either ANSI format, however, the codes
                    'A', 'P', and 'N' are used for am, pm, and
                    noon, respectively.
                    (5:00:00pm/17:00:00) }
  TF_TimeZone    = #010000; { If set output the time zone as -zzz
                    after the time else omit it
                    (17:00:00- EDT/17:00:00) }

  TF_NoColumns   = #040000; { If set output numeric date/time
                    quantities in the smallest fields
                    into which they will fit, else
                    output them in fixed size fields. If
                    not set, the date/time will be
                    output in fixed length fields, thus
                    making this format appropriate for
                    columnar display. Note that TF_FullMonth
                    and TF_FullWeekday are currently NOT padded
                    with blanks, even if TF_NoColumns is off. }
  TF_BlankPad    = #020000; { If set, pad fixed-width numbers with blanks
                    instead of zeroes WHERE REASONABLE. Ignored
                    if TF_NoColumns is set. }

  TF_Never      = #100000; { If set allow the distinguished time
```

```
                    value NEVER to be output as the
                    string 'Never' else signal an
                    error }
```

The following flags are returned to indicate which fields of the date and time were present upon parsing a date/time string.

```
const
  TP_Weekday    = #000001; { Weekday present }
  TP_Date       = #000002; { Date present }
  TP_Time       = #000004; { Time present }
  TP_Zone       = #000010; { TimeZone present }
  TP_Never      = #000020; { Time input was NEVER }
  TP_RESERVED   = #177740; { Reserved for expansion }
```

String_255: The maximum length string. We parse dates from such strings. This definition should probably be somewhere else.

```
type
  String_255    = string[255];
```

8.3 Exceptions

BadDateTime	This exception is raised if a bad date/time value is passed to any of the TimeServer routines, or if the TimeFormat flags (for conversion to String format) are invalid.
TimeNotInitialized	This exception is raised if the system date, time, and time zone have not been set.

8.4 Procedures

The following procedures are from TimeUser.pas.

SetDateTime

Sets current date and time.

Call:

```
Procedure SetDateTime( ServPort : port;
                       ITime    : Internal_Time)
```

Parameters:

ServPort–TimePort (service port to Time Server)

ITime–Internal_Time record for time to set.

Returns:

Raises TimeNotInitialized if system time zone and daylight switch have not been set.

SetSystemZone

Sets the system defaults for time zone and whether to use Daylight time.

Call:

```
Procedure SetSystemZone( ServPort : port;
                TimeZone : integer;
                DSTWhenTimely : boolean)
```

Parameters:

ServPort–TimePort (service port to Time Server)

TimeZone–System time zone to set (refer to the constant definitions).

DSTWhenTimely–Whether to use daylight time during the USA daylight time interval.

GetDateTime

Gets current time, in Internal_Time format.

Call:

```
function GetDateTime (ServPort : Port)
      : Internal_Time
```

Parameters:

ServPort–TimePort (service port to TimeServer)

Returns:

Raises TimeNotInitialized if system time has not been set.

GetUserTime

Gets current time, in User_Time format, according to system time zone and daylight time defaults.

Call:

```
function GetUserTime ( ServPort : Port)
          : User_Time
```

Parameters:

ServPort–TimePort (service port to Time Server)

Returns:

Raises TimeNotInitialized if system time has not been set.

```
function GetStringTime( ServPort  :  Port;
                        TimeFormat :  integer)
                                 : String
```

Gets current time, in string format.

Call:

GetStringTime

Parameters:

ServPort–TimePort (service port to Time Server)

TimeFormat–Format for the string (refer to the constant definitions)

Returns:

Raises TimeNotInitialized if system time has not been set.

T_IntToZone

Converts internal time to user time, according to supplied time zone.

Call:

```
function T_IntToZone ( ServPort  : port;
                       ITime     : Internal_Time;
                       WantZone  : Zone_Info)
          : User_Time
```

Parameters:

ServPort–TimePort (service port to Time Server)

ITime–Internal_Time record

WantZone–Zone_Info record containing Zone and daylight desired.

Returns:

User_Time record representing ITime.

T_IntToUser

Converts internal time to user time, according to supplied time zone and daylight time defaults.

Call:

```
function T_IntToUser ( ServPort  :  port;
                       ITime     :  Internal_Time)
        : User_Time
```

Parameters:

ServPort–TimePort (service port to Time Server)

ITime–Internal_Time record

Returns:

User_Time record representing ITime.

T_UserToInt

Converts user time value to internal time.

Call:

```
function T_UserToInt ( ServPort  :  port;
                       UTime     :  User_Time)
        : Internal_Time
```

Parameters:

ServPort–TimePort (service port to Time Server)

UTime–User_Time record

Returns:

Internal_Time corresponding to UTime.

T_UserToString

Converts a user time record to a string representing the time, according to the conversion parameters.

Call:

```
function T_UserToString ( ServPort  :  port;
                          UTime     :  User_Time
                          TimeFormat:  integer)
        : String
```

Parameters:

ServPort–TimePort (service port to Time Server)

UTime–User_Time record

TimeFormat–Format desired for the output (refer to the constant definitions)

Returns:

A string representation of UTime.

T_IntToString

Converts an internal time record to a string representing the time, according to the conversion parameters. The system defaults for time zone and daylight time are used.

Call:

```
function T_IntToString ( ServPort  :  port;
                         ITime     :  Internal_Time
                         TimeFormat:  Integer)
           : String
```

Parameters:

ServPort–TimePort (service port to Time Server)

ITime–Internal_Time record

TimeFormat–Format desired for the output (refer to the constant definitions.)

Returns:

A string representation of ITime.

T_StringToUser

Converts a string to a user-time record.

Call:

```
function T_StringToUser( ServPort  : port;
                         STime     : String_255;
                       var Index   : integer;
                       var WhatIFound: integer)
         : User_Time
```

Parameters:

ServPort–TimePort (service port to Time Server)

STime–String to be converted to time.

Index–Position in string to start scanning for time. Returns the first character past the end of the valid time string.

WhatIFound–Returns what was parsed from the time string .

Returns:

Raises BadDateTime if STime malformed.

T_StringToInt

Converts a string to an internal-time record.

Call:

```
function T_StringToInt ( ServPort  : port;
                         STime     : String_255;
                       var Index   : integer;
                       var WhatIFound: integer)
            : Internal_Time
```

Parameters:

ServPort–TimePort (service port to Time Server)

STime–String to be converted to time.

Index–Position in string to start scanning for time. Returns the first character past the end of the valid time string.

WhatIFound–Returns what was parsed from the time string

Returns:

Raises BadDateTime if STime malformed.

T_Never

Returns the internal time value representing "never".

Call:

```
function T_Never ( ServPort  :  Port)
         : Internal_Time
```

Parameters:

ServPort–TimePort (service port to Time Server)

Returns:

Internal_Time record representing Never.

8.5 Valid Time Zones

The valid time zones accepted by T_StringToUser and T_StringToInt are,

GMT	Greenwich MeanTime	
UT	UniversalTime (Same as GMT)	
NST	Newfoundland Standard Time	(-3:30)
AST	Atlantic Standard Time	(-4 hours)
ADT	Atlantic Daylight Time	
EST	Eastern Standard Time	(-5 hours)
EDT	Eastern Daylight Time	
CST	Central Standard Time	(-6 hours)
CDT	Central Daylight Time	
MST	Mountain Standard Time	(-7 hours)
MDT	Mountain Daylight Time	
PST	Pacific Standard Time	(-8 hours)
PDT	Pacific Daylight Time	
YST	Yukon Standard Time	(-9 hours)
YDT	Yukon Daylight Time	
HST	Hawaii-Alaska Standard Time	(-10 hours)
HDT	Hawaii-Alaska Daylight Time	
BST	Bering Standard Time	(-11 hours)
BDT	Bering Daylight Time	

9 The Typescript Manager

9.1 Introduction

The Typescript manager maintains standard text windows (Typescripts), providing line editing and redisplay functions for user programs that do not require graphics output or elaborate input control.

Typescript remembers the last several pages (a window's worth) of text output by the program using a typescript. When a text window changes state, size, or coveredness, Typescript will also redisplay information that may have scrolled from the window.

Typescript allows the user to edit input lines using a subset of the commands available in the system editor. The user can use all of the editor's single- line editing functions, and recall previous input lines to be edited into new input lines.

Typescript also handles Escape Completion for each typescript. The user may type a partial filename and press the Escape key; Typescript will ask the file system to complete the file name by finding the longest unambiguous match to the partial name. It will then add the name to the line of input, so that the user can specify the rest of the name or add more to the line.

Typescript can either stop at the end of each screenful of output ('more' mode) or scroll output continuously. The user can select which mode to use, under keyboard control.

In 'More' mode, when a full page of output has been displayed, a black bar appears at the bottom of the window. The user then presses LineFeed to display the next page of output.

In continuous scroll mode, the user can use the Process Control Functions Suspend and Resume to stop or start output.

9.2 Use

Each typescript maintained by the Typescript manager has its own port. All requests for input, output, and control of a typescript are directed to its port. There is a master typescript port used to create new typescripts.

When a user program is started, the mater typescript port is TypescriptPort and the port associated with the program's window is UserTypescript. Both of these are in PascalInit.Pas. Pascal input and output through the default files INPUT and OUTPUT is directed to UserTypescript (unless input or output has been redirected).

A program may create a new Typescript in an existing window by using the call STSOpenWindow and providing it with the window to use. It returns a port for the typescript in that window.

The program may then request a full line of input with STSGetString. This will allow the user to type a line of input, using all of the line editing commands. When the user types RETURN, the line is finished and is returned to the program.

A program may ouput a line of text by calling STSPutString, supplying the typescript and the string to output. Each LineFeed chatacter in the string ends a line of text, scrolling the typescript and putting the ntext character at the start of the next line. The string does not need to end with a line feed.

A program can ask the Tyepscript Manager to stop refreshing the screen. The call STSGrabWindow returns the window associated with the supplied typescript. When the program is terminated, the Typesscript Manager will resume control over the window and refresh it, erasing whatever the program had displayed on it.

Since each Typescrip may be used by a program or shell that has its own environment, a program can specify the environment connection to be used by the typescript. This controls the list of files scanned for Escape Completion. Each Typescript has its own Environment Connection.

9.3 Definitions

The following definitions are from TSDefs.pas.

```
type

 Typescript = Port;
 TString255 = String[255];

 TSCharArray = packed array[0 .. 1] of Char;
 pTSCharArray = ↑ TSCharArray;
```

9.4 Routines

The following routines are from TSUser.pas.

STSOpen

Creates a Typescript inside a viewport. The typescript uses the system font, displays long lines by wrapping around, and stores three windows' worth of output.

Call:

```
Function STSOpen(
        ServPort: Port;
        vp :      viewport;
        env:      Port)
    : Typescript
```

Parameters:

ServPort–The master Typescript service port (TypescriptPort)

vp–The viewport to contain the typescript.

env–An Environment Manager connection, used to define the environment for Escape Completion on this typescript.

Returns:

A port for a new typescript.

STSOpenWindow

Creates a Typescript inside a window. The typescript uses the system font, displays long lines by wrapping around, and stores three windows' worth of output.

Call:

```
Function STSOpen(
        ServPort:  Port;
        w :      window;
        env:      Port)
    : Typescript
```

Parameters:

ServPort–The master Typescript service port (TypescriptPort)

w–The window to contain the typescript.

env–An Environment Manager connection, used to define the environment for Escape Completion on this typescript.

Returns:

A port for a new typescript.

STSFullOpen

Creates a Typescript inside a viewport, allowing the program to select more parameters for the typescript.

Call:

```
Function STSFullOpen(
        ServPort:  Port;
        vp  :    viewport;
        env:      Port;
        fontName : TString255;
        doWrap  :  Boolean;
        dispPages : Integer)
    : Typescript
```

Parameters:

ServPort–The master Typescript service port (TypescriptPort)

vp–The viewport to contain the typescript.

env–An Environment Manager connection, used to define the environment for Escape Completion on this typescript.

fontName–The name of the font containing the font to use for the typescript. It must be an absolute path name.

doWrap–if TRUE, displays lines wider than the window by wrapping to the start of the next line. If FALSE, truncates long lines at the right margin of the window.

Returns:

A port for a new typescript.

STSFullOpenWindow

Creates a Typescript inside a window, allowing the program to select more parameters for the typescript.

Call:

```
Function STSFullOpen(
         ServPort:  Port;
         w :        Window;
         env:       Port;
         fontName : TString255;
         doWrap  :  Boolean;
         dispPages : Integer)
    : Typescript
```

Parameters:

ServPort–The master Typescript service port (TypescriptPort)

w–The window to contain the typescript.

env–An Environment Manager connection, used to define the environment for Escape Completion on this typescript.

fontName–The name of the font containing the font to use for the typescript. It must be an absolute path name.

doWrap–if TRUE, displays lines wider than the window by wrapping to the start of the next line. If FALSE, truncates long lines at the right margin of the window.

Returns:

A port for a new typescript.

STSGetChar

Returns a singel character of input from a typescript. If a full line has not been typed, it invokes the line editor and waits until a line is completed. If a line has been typed, it removes the next character from the start of the line and returns it.

Call:

```
Function STSGetChar(
        ServPort:  Typescript)
  : Char
```

Parameters:

ServPort–Port for the typescript.

Returns:

The first character on the input line.

STSGetString

Returns an entire line of input from a typescript. If a full line has not been typed, it invokes the line editor and waits until a line is completed. It then returns the entire line.

Call:

```
function STSGetString(
         ServPort  : Typescript)
    : TString255
```

Parameters:

ServPort–Port for the typescript.

Returns:

The entire input line.

STSPutChar

Writes a single character to a typescript.

Call:

```
Procedure STSPutChar(
        ServPort  : Typescript;
        ch        : Char)
```

Parameters:

ServPort–The port for the typescript.

ch–The single character to output.

A LineFeed character ends the current output line. A BELL character (control G) flashes the viewport containing the typescript. Any other character is appended to the current line, possibly translated:

Control character (chr(0)..chr(13)) are displayed as ↑Char.

DEL is displayed as ↑{.

Printing characters (space through '}') are displayed as themselves.

Any character greater than chr(127) is displayed as the corresponding character in the font for the typescript, minus 128. This allows character in the font with numeric values less than 32 to be displayed as normal printing characters.

STSPutString

Writes a string of characters to a typescript. Each character in the string is displayed according to the description for STSPutChar.

Call:

```
Procedure STSPutString(
          ServPort : Typescript;
          s        : TString255)
```

Parameters:

ServPort–The port for the typescript.

s–The string to output.

STSFlushInput

Flushes any partially entered input line from a typescript.

Call:

```
Procedure STSFlushInput(
          ServPort : Typescript)
```

Parameters:

ServPort–The port for the typescript.

STSFlushOutput

Forces any queued output for a Typescript to be displayed on the screen. STSPutChar and STSPutString may not display the output characters immediately, giving strange results if one tries to use the same viewport for simple text output and for graphics. STSFlushOutput ensures that all Typescript output is displayed on the screen before it returns.

Call:

```
Procedure STSFlushOutput(
        ServPort : Typescript)
```

Parameters:

ServPort–The port for the typescript.

STSChangeEnv

Changes the Environment connection associated with a typescript. The Environment connection determines the searchlists used for Escape Completion within that typescript.

Parameters:

ServPort–The port for the typescript.

env–The new Environment Connection to use.

STSGrabWindow

STSGrabWindow tells Typescript to stop monitoring the state of the window containing a Typescript for change is state, size or coveredness. A program such as the editor uses this procedure to gain control of the default user window to use it for graphics. When the program terminates, typescript regains control of the window and redisplays its contents as of the time STSGrabWindow was called; any changes that the program made to the window are lost.

Call:

```
Function STSGrabWindow(
        ServPort : Typescript;
        kPort :    Port)
  : Window
```

Parameters:

ServPort–The port for the typescript.

kPort–A port that the user program has ownership
rights for or that will otherwise be deallocated when the

program terminates. Typescript regains control of the window when this port is deleted.

Returns:

The window that the Typescript is using.